HED Resources

Release 0.0.1

HED Working Group

Jul 06, 2023

OVERVIEW:

1 What is HED? **3**

2 How is HED used? **5**

3 HED and BIDS **7**

4 HED Tools **9**

5 Where to begin? **11**

6 History and Support **13**

6.1 Introduction to HED 13

6.1.1 Brief history of HED 13

6.1.2 Goals of HED 14

6.1.3 A basic HED annotation 14

6.1.4 How to get started 15

6.2 What's new? 15

6.3 How do you use HED? 17

6.3.1 As an experimenter 18

6.3.1.1 Planning and running an experiment 18

6.3.1.2 Post-processing the event data 19

6.3.2 As a data annotator 20

6.3.2.1 Standardizing the format 21

6.3.2.2 Adding HED annotations 22

6.3.2.3 Checking correctness 23

6.3.3 As a data analyst 23

6.3.3.1 Understanding the data 24

6.3.3.2 Preparing the data 24

6.3.3.3 Analyzing the data 24

6.3.4 As a tool developer 25

6.3.4.1 Integration with existing tools 26

6.3.4.2 The HED code base 26

6.3.4.3 Future development plans 27

6.3.5 As a schema builder 27

6.3.5.1 Viewing available schemas 28

6.3.5.2 Improving an existing schema 28

6.3.5.3 Creating a new library schema 28

6.3.5.4 Private vocabularies and extensions 29

6.4 BIDS annotation quickstart 29

6.4.1 How HED works in BIDS 30

6.4.1.1 BIDS event files 30

6.4.1.2 JSON event sidecars . . . 31
6.4.2 Create a JSON template . . . 32
6.4.2.1 Step 1: Select generate JSON . . . 33
6.4.2.2 Step 2: Upload an events file. . . . 34
6.4.2.3 Step 3: Select columns to annotate . . . 35
6.4.2.4 Step 4: Download the template. . . . 37
6.4.2.5 Step 5: Complete the annotation. . . . 38
6.4.3 Spreadsheet templates . . . 39
6.4.3.1 Step 1: Select extract HED spreadsheet . . . 39
6.4.3.2 Step 2: Upload a sidecar and extract. . . . 40
6.4.3.3 Step 3: Edit the spreadsheet . . . 41
6.4.3.4 Step 4: Merge the spreadsheet . . . 41
6.5 HED annotation quickstart . . . 43
6.5.1 What is HED annotation? . . . 43
6.5.2 A recipe for simple annotation . . . 43
6.6 HED validation guide . . . 46
6.6.1 What is HED validation? . . . 46
6.6.2 Types of errors . . . 47
6.6.3 Available validators . . . 47
6.6.3.1 Python validator . . . 47
6.6.3.2 JavaScript validator . . . 47
6.6.3.3 MATLAB support . . . 47
6.6.4 Validation strategies . . . 48
6.6.4.1 Validation in BIDS . . . 48
6.6.4.2 HED online validation . . . 49
6.6.4.3 Validation for MATLAB users . . . 50
6.6.4.4 Validation for Python users . . . 50
6.7 HED search guide . . . 52
6.7.1 HED search basics . . . 53
6.7.1.1 Calling syntax . . . 53
6.7.1.2 Single tag queries . . . 54
6.7.1.3 Logical queries . . . 55
6.7.1.4 Group queries . . . 55
6.7.2 Where can HED search be used? . . . 56
6.8 HED summary guide . . . 56
6.8.1 Column value summary . . . 57
6.8.2 HED tag summary . . . 58
6.8.3 Experimental design summary . . . 59
6.9 HED conditions and design matrices . . . 61
6.9.1 HED annotations for conditions . . . 61
6.9.1.1 Direct condition variables . . . 62
6.9.1.2 Defined condition variables . . . 64
6.9.1.3 Direct vs defined approaches . . . 65
6.9.1.4 Column vs row annotations . . . 65
6.9.2 Experimental design concepts . . . 69
6.9.2.1 Design matrices and factor variables . . . 70
6.9.2.2 Types of condition encoding . . . 70
6.10 File remodeling quickstart . . . 71
6.10.1 What is remodeling? . . . 71
6.10.2 The remodeling process . . . 74
6.10.3 JSON remodeling files . . . 74
6.10.3.1 Basic remodel operation syntax . . . 75
6.10.3.2 Applying multiple remodel operations . . . 75
6.10.3.3 More complex remodeling . . . 76

6.10.3.4 Remodeling file locations 78
6.10.4 Using the remodeling tools 78
6.10.4.1 Online tools for debugging 78
6.10.4.2 The command-line interface 80
6.10.4.3 Jupyter notebooks for remodeling 81
6.11 HED schema developer's guide 81
6.11.1 Setting up for schema development 81
6.11.2 Design principles for schema 82
6.11.3 Defining a schema 82
6.11.4 Schema namespaces 83
6.11.5 Attributes and classes 83
6.11.5.1 Required sections 83
6.11.5.2 Relation to base schema 84
6.11.5.3 Schema properties 84
6.11.5.4 Unit classes 84
6.11.5.5 Value classes 84
6.11.5.6 Schema attributes 84
6.11.5.7 Syntax checking 84
6.11.5.8 Procedure for updating a schema. 85
6.11.6 HED schema details 85
6.11.7 Further documentation 85
6.12 HED online tools 85
6.12.1 Browser-based access 86
6.12.1.1 Events files 86
6.12.1.2 Sidecar files 88
6.12.1.3 Spreadsheet files 91
6.12.1.4 String online tools 93
6.12.1.5 Schema online tools 94
6.12.2 HED RESTful services 95
6.12.2.1 Service setup 95
6.12.2.2 Request format 96
6.12.2.3 Service responses 98
6.13 CTagger GUI tagging tool 99
6.13.1 CTAGGER installation 99
6.13.1.1 CTAGGER standalone installation 99
6.13.1.2 CTAGGER in EEGLAB 99
6.13.2 Loading BIDS event files 99
6.13.3 Adding HED annotation 101
6.13.3.1 Validating your annotation 102
6.14 File remodeling tools 103
6.14.1 Overview of remodeling 104
6.14.1.1 Transformation operations 104
6.14.1.2 Summarization operations 105
6.14.1.3 Available operations 105
6.14.2 Installing the remodel tools 107
6.14.3 Remodel command-line interface 108
6.14.3.1 Calling remodel tools 108
6.14.3.2 Remodel command-line arguments 108
6.14.4 Remodel scripts 110
6.14.4.1 Backing up files 110
6.14.4.2 Remodeling files 112
6.14.4.3 Restoring files 113
6.14.5 Remodel with HED 114
6.14.5.1 Extracting HED information from BIDS 114

6.14.5.2 Directly specifying HED information . . . 114
6.14.6 Remodel error handling . . . 115
6.14.6.1 Errors in the remodel file . . . 115
6.14.6.2 Execution-time remodel errors . . . 115
6.14.7 Remodel sample files . . . 116
6.14.7.1 Sample remodel file . . . 116
6.14.7.2 Sample remodel event file . . . 116
6.14.7.3 Sample remodel sidecar file . . . 117
6.14.8 Remodel transformations . . . 118
6.14.8.1 Factor column . . . 118
6.14.8.2 Factor HED tags . . . 119
6.14.8.3 Factor HED type . . . 120
6.14.8.4 Merge consecutive . . . 122
6.14.8.5 Remap columns . . . 124
6.14.8.6 Remove columns . . . 126
6.14.8.7 Remove rows . . . 127
6.14.8.8 Rename columns . . . 128
6.14.8.9 Reorder columns . . . 129
6.14.8.10 Split rows . . . 130
6.14.9 Remodel summarizations . . . 132
6.14.9.1 Summarize column names . . . 133
6.14.9.2 Summarize column values . . . 134
6.14.9.3 Summarize definitions . . . 136
6.14.9.4 Summarize HED tags . . . 139
6.14.9.5 Summarize HED type . . . 142
6.14.9.6 Summarize HED validation . . . 143
6.14.9.7 Summarize sidecar from events . . . 145
6.14.10 Remodel implementation . . . 147
6.14.10.1 The PARAMS dictionary . . . 147
6.14.10.2 Operation class constructor . . . 148
6.14.10.3 The do_op implementation . . . 148
6.14.10.4 The do_op for summarization . . . 149
6.14.10.5 Additional requirements for summarization . . . 149
6.15 HED Python tools . . . 150
6.15.1 Jupyter notebooks for HED in BIDS . . . 150
6.15.1.1 Summarize BIDS event files . . . 150
6.15.1.2 Extract a JSON sidecar template . . . 151
6.15.1.3 JSON sidecar to spreadsheet . . . 151
6.15.1.4 Validate HED in a BIDS dataset . . . 152
6.15.2 Jupyter notebooks for data curation . . . 152
6.15.2.1 Consistency of BIDS event files . . . 152
6.15.3 Calling HED tools . . . 153
6.15.3.1 Getting a list of files . . . 153
6.15.3.2 Dictionaries of filenames . . . 153
6.15.3.3 Logging processing steps . . . 155
6.16 HED JavaScript tools . . . 156
6.16.1 Javascript tool installation . . . 156
6.16.2 Javascript package organization . . . 156
6.16.3 Javascript programmatic interface . . . 156
6.17 HED MATLAB tools . . . 157
6.17.1 HED services in MATLAB . . . 157
6.17.1.1 Overview of service requests . . . 158
6.17.1.2 Setting up a session from MATLAB . . . 158
6.17.1.3 Creating a request structure . . . 159

6.17.1.4 Making a service request . . . 160
6.17.1.5 Decoding a service response . . . 160
6.17.2 EEGLAB plug-in integration . . . 161
6.17.2.1 Installing *HEDTools* . . . 161
6.17.2.2 Annotating datasets . . . 162
6.17.2.3 HED-based epoching . . . 166
6.17.3 Python HEDTools in MATLAB . . . 167
6.17.3.1 Getting started . . . 167
6.17.3.2 MATLAB wrappers for HEDTools . . . 170
6.17.3.3 MATLAB functions for Python . . . 171
6.18 HED schemas . . . 171
6.18.1 HED schema basics . . . 171
6.18.1.1 Tag forms . . . 172
6.18.1.2 Types of schemas . . . 172
6.18.2 Viewing schemas . . . 173
6.18.3 Available schemas . . . 173
6.18.3.1 The standard schema . . . 173
6.18.3.2 The SCORE library . . . 173
6.18.3.3 The LISA library . . . 173
6.19 HED test datasets . . . 174
6.19.1 eeg_ds002893s_hed . . . 175
6.19.2 eeg_ds003645s_hed . . . 175
6.19.3 eeg_ds003645s_hed_column . . . 175
6.19.4 eeg_ds003645s_hed_inheritance . . . 175
6.19.5 eeg_ds003645s_hed_library . . . 175
6.19.6 eeg_ds003645s_hed_longform . . . 175
6.19.7 eeg_ds004105s_hed . . . 175
6.19.8 eeg_ds004106s_hed . . . 176
6.19.9 eeg_ds004117s_hed_sternberg . . . 176
6.19.10 fmri_ds002790s_hed_aomic . . . 176
6.19.11 fmri_soccer21_hed . . . 176
6.19.12 BIDS validation . . . 176

7 Indices and tables **177**

Links

- PDF docs
- Source code

CHAPTER ONE

WHAT IS HED?

HED (**'Hierarchical Event Descriptors'**, pronounced either as /hed/ or /H//E//D/) is a framework for using a controlled yet extensible vocabulary to systematically describe experiment events of all types (perceptual, action, experiment control, task . . .).

The **goals of HED** are to enable and support its users to **store and share** recorded data in a fully **analysis-ready** format, and to support efficient (and/or extended cross-study) data **search and analysis**.

CHAPTER

TWO

HOW IS HED USED?

HED enables users to use a standard method to **detail the nature** of each experiment event, and to record information about **experiment organization**, thus creating a permanent, both human- and machine-readable record embedded in the data record for use in any further analysis, re-analysis, and meta/mega-analysis.

HED may be used to **annotate any type of data** – but particularly data acquired in functional brain imaging (EEG, MEG, fNIRS, fMRI), multimodal (aka MoBI, mobile brain/body imaging), psychophysiological (ECG, EMG, GSR), or purely behavioral experiments.

HED annotations are composed of comma-separated **tags** from a hierarchically-structured vocabulary called the HED standard schema (possibly augmented by terms from one or more specialized **HED library schemas**).

HED library schemas for use in individual research subfields as well as the standard schema and vocabularies under development are housed in the hed-schemas.

The **HED working group** is an ongoing open-source development organization whose mission is to extend and maintain the HED standard and associated tools. Visit the hed-standard site on GitHub for information on how to join the HED community of users and developers.

CHAPTER

THREE

HED AND BIDS

HED was accepted (2019) into the top-level BIDS (Brain Imaging Data Structure) standard, thus becoming an integral part of the BIDS data storage standards for an ever-increasing number of neuroimaging data modalities.

An efficient approach to integrating HED event descriptions into BIDS metadata has been demonstrated in this 2021 paper.

CHAPTER FOUR

HED TOOLS

Currently, tools using HED for data annotation, validation, search, and extraction are available for use online, or (as MATLAB functions) within the EEGLAB environment running on Matlab.

CHAPTER

FIVE

WHERE TO BEGIN?

To begin using HED tools to tag, search, and analyze data, browse the HED resources page. Visit the How can you use HED? guide for information about how specific types of users can leverage HED.

CHAPTER

SIX

HISTORY AND SUPPORT

HED (Gen 1, version < 4.0.0) was first proposed and developed by Nima Bigdely-Shamlo within the HeadIT project at the Swartz Center for Computational Neuroscience (SCCN) of the University of California San Diego (UCSD) under funding by The Swartz Foundation and by U.S. National Institutes of Health (NIH) grants R01-MH084819 (Makeig, Grethe PIs) and R01-NS047293 (Makeig PI).

Further **HED (Gen 2, 4.0.0 <= version < 8.0.0)** development led by Kay Robbins of the University of Texas San Antonio was funded by The Cognition and Neuroergonomics Collaborative Technology Alliance (CaN CTA) program of U.S Army Research Laboratory (ARL) under Cooperative Agreement Number W911NF-10-2-0022.

HED (Gen 3, version >= 8.0.0) is now maintained and further developed by the HED Working Group led by Kay Robbins and Scott Makeig with Dung Truong, Monique Denissen, Dora Hermes Miller, Tal Pal Attia, and Arnaud Delorme, with funding from NIH grant RF1-MH126700.

HED is an open research community effort; others interested are invited to participate and contribute. Visit the HED project homepage for links to the latest developments..

6.1 Introduction to HED

HED (an acronym for Hierarchical Event Descriptors) is an evolving framework and structured vocabulary for annotating data, particularly data events to enable data search, extraction, and analysis. Specifically, the goal of HED is to allow researchers to annotate what happened during an experiment, including experimental stimuli and other sensory events, participant responses and actions, experimental design, the role of events in the task, and the temporal structure of the experiment.

The resulting annotation is **machine-actionable**, meaning that it can be used as input to algorithms without manual intervention. HED facilitates detailed comparisons of data across studies and promotes accurate interpretation of what happened as an experiment unfolds.

6.1.1 Brief history of HED

HED was originally proposed by Nima Bigdely-Shamlo in 2010 to support annotation in HeadIT, an early public repository for EEG data hosted by the Swartz Center for Computational Neuroscience, UCSD (Bigdely-Shamlo et al. 2013). HED has undergone several revisions and substantial infrastructure development since that time.

The BIDS (Brain Imaging Data Structure) standards group incorporated HED as annotation mechanism in 2019. In 2019, work also began on a rethinking of the HED vocabulary design, resulting in the release of the third generation of HED in August 2021, representing a dramatic increase in annotation capacity and a significant simplification of the user experience.

New in HED (versions 8.0.0+) released August 2021.

1. Improved vocabulary structure
2. Short-form annotation
3. Library schema
4. Definitions
5. Temporal scope
6. Encoding of experimental design

See the HED Specification and the **documentation summary** for additional details.

6.1.2 Goals of HED

Event annotation documents the things happening during data recording regardless of relevance to data analysis and interpretation. Commonly recorded events in electrophysiological data collection include the initiation, termination, or other features of **sensory presentations** and **participant actions**. Other events may be **unplanned environmental events** (for example, sudden onset of noise and vibration from construction work unrelated to the experiment, or a laboratory device malfunction), events recording **changes in experiment control** parameters as well as **data feature events** and control **mishap events** that cause operation to fall outside of normal experiment parameters. The goals of HED are to provide a standardized annotation and supporting infrastructure.

Goals of HED.

1. **Document the exact nature of events** (sensory, behavioral, environmental, and other) that occur during recorded time series data in order to inform data analysis and interpretation.
2. **Describe the design of the experiment** including participant task(s).
3. **Relate event occurrences** both to the experiment design and to participant tasks and experience.
4. **Provide basic infrastructure** for building and using machine-actionable tools to systematically analyze data associated with recorded events in and across data sets, studies, paradigms, and modalities.

Current systems in neuroimaging experiments do not record events beyond simple numerical (3) or text (Event type Target) labels whose more complete and precise meanings are known only to the experimenter(s).

A central goal of HED is to enable building of archives of brain imaging data in a amenable to large scale analysis, both within and across studies. Such event-related analysis requires that the nature(s) of the recorded events be specified in a common language.

The HED project seeks to formalize the development of this language, to develop and distribute tools that maximize its ease of use, and to inform new and existing researchers of its purpose and value.

6.1.3 A basic HED annotation

HED annotations are comma-separated lists of tags selected from a hierarchically-organized vocabulary.

A simple HED annotation of presentation of a face image stimulus.

Sensory-event, Experimental-stimulus, (Visual-presentation, (Image, Face, Hair)), (Image, Pathname/f032.bmp), Condition-variable/Famous-face, Condition-variable/Immediate-repeat

The annotation above is a very basic annotation of an event marker representing the presentation of a face image with hair. The event marker represents an experimental stimulus with two experimental conditions *Famous-face* and *Immediate-repeat* in effect.

Because HED has a structured vocabulary, other researchers use the same terms, making it easier to compare experiments. Further, the HED infrastructure supports associating of these annotation strings with the actual event markers during processing, allowing tools to locate event markers using experiment-independent strategies.

The annotation in the example uses the most basic strategy for annotating condition variables — just naming the different conditions. However, even this simple strategy allows tools to distinguish among events taken under different task conditions. HED also provides more advanced strategies that allow downstream tools to automatically extract dataset-independent design matrices.

Every term in the HED structured vocabulary (HED schema) must be unique, allowing users to use a single word for each annotation tag. Tools can expand into their full paths within the HED schema, allowing tools to leverage hierarchical relationships during searching.

An equivalent long-form HED annotation of face image stimulus from above.

Event/Sensory-event,
Property/Task-property/Task-event-role/Experimental-stimulus,
(Property/Sensory-property/Sensory-presentation/Visual-presentation,
(Item/Object/Man-made-object/Media/Visualization/Image,
Item/Biological-item/Anatomical-item/Body-part/Head/Face,
Item/Biological-item/Anatomical-item/Body-part/Head/Hair)),
(Item/Object/Man-made-object/Media/Visualization/Image,
Property/Informational-property/Metadata/Pathname/f032.bmp),
Property/Organizational-property/Condition-variable/Famous-face,
Property/Organizational-property/Condition-variable/Immediate-repeat

HED is also extensible, in that most nodes can be extended to include more specific terms. HED also permits **library schema**, which are specialized vocabularies. HED tools support seamless annotations that include both terms from the base schema and from specialized, discipline-specific vocabularies.

6.1.4 How to get started

The ***HED annotation quickstart*** provides a simple step-by-step guide to doing basic HED annotation, while the ***Bids annotation quickstart*** introduces the various types of annotation that should be included in a BIDS (**Brain Imaging Data Structure**) dataset. This tutorial also includes instructions for using the online tools to start the annotation process.

6.2 What's new?

July 5, 2023: **HEDTools version 0.3.1 released**

HEDTools version 0.3.1 has been released on PyPI.
See also **hed-python** on GitHub.
This patch includes improvement to models exception handling, minor bug fixes, and improvement of format for JSON format of remodeling summaries.

June 20, 2023: **HEDTools version 0.3.0 released**

HEDTools version 0.3.0 has been released on PyPI.
See also hed-python on GitHub.
This version uses the new DataFrame implementation of the models for improved efficiency.

June 14, 2023: **HED brain initiative meeting video poster online.**

HED video poster available until June 2024:
HED: Annotation standards and software infrastructure to enable sharing of analysis-reading neuroimaging and behavior data
Presented as part of the Brain Initiative Meeting 2023.

June 2, 2023: **HED online tools have a new look.**

The HED online tools at https://hedtools.ucsd.edu/hed have been redesigned and streamlined.

May 12, 2023: **Version 3.2.0 of the HED specification released.**

Version 3.2.0 of the HED specification introduces the `Inset` tag as well as partnered and rooted library schemas. This version is the first to support the curly brace notation in sidecars.

April 28, 2023: **HED standard schema v8.2.0 released.**

The HED schema v8.2.0 has just been released. This release supports the `Inset` tag for annotating intermediate points in an ongoing event as well as partnered library schemas.

April 25, 2023: **HED playlist goes live on YouTube.**

The Hierarchical Event Descriptor playlist is now available on YouTube. A 5-part short-course on HED (from OHBM 2022) has just been released.

April 6, 2023: **Version 3.1.0 of the HED specification released.**

Version 3.1.0 of the HED specification clarifies existing features.

HED validators will be able to link error messages directly to descriptions in the spec.

April 3, 2023: **New versions of HED schema browser available.**

You can now use a single HED Schema Browser to view both standard and library schemas.

A prelease viewer is available for viewing all prerelease HED schemas.

March 28, 2023: **Release of HED Javascript validator for BIDS v3.9.0 on npm**

The new version supports checking of onsets and offsets.

March 27, 2023: **HED Workshop at CNS 2023**

Title: *Recording what happened during your experiment using Hierarchical Event Descriptors (HED)*
Presenter: Scott Makeig UCSD and assisted by members of the HED Working Group.
Time and location: 12:15-1:15 pm Seacliff Room Hyatt Regency San Francisco Hotel.

March 24, 2023: **HED is now on Twitter**

Follow us at @HedDescriptors

March 22-23, 2023: **HED at the 2023 Annual Assembly of the Global Brain Consortium**

Directions in EEG data-sharing - panel co-chaired by Kay Robbins and Dora Hermes
Program and Registration

Feb 28, 2023: **INCF Software Highlight on HED**

Title: *HED, a practical system for describing an experiment using an analysis-ready framework*
Presenter: Kay Robbins UTSA, member of the HED Working Group.
Abstract.

February 22, 2023: **HED YouTube Channel goes live**

First video: **HED Tagging 101**
HED YouTube channel: **https://www.youtube.com/@hedworkinggroup/videos**

February 14, 2023: **HEDTools 0.2.0 is released.**

- This release includes the HED remodeling tools.
- Improved local caching and schema validation messages are included.
- This is the first release with distinct `stable`, `master`, and `develop` branches.

January 28, 2023: **HED SCORE Library v1.0.0 released**

The HED score library schema v1.0.0 has been officially released. This library is based on the Standardized Computer-based Organized Reporting of EEG (**SCORE**) standard and adapted for annotation using the HED infrastructure and requirements. For more information see the **SCORE schema library** guide.

January 20, 2023: **New preprint available**

Actionable event annotation and analysis in fMRI **preprint** *Monique Denissen*, *Fabio Richlan*, *Jürgen Birkibauer*, *Mateusz Pawlik*, *Anna Ravenschlag*, *Nicole Himmelstoß*, *Florian Hutzler*, *Kay Robbins* Supporting materials and data are available at **https://osf.io/93km8/**. Chapter to appear in the book *Methods for analyzing large neuroimaging datasets* edited by Robert Whelan and Herve Lemaitre.

January 4, 2023: **New preprint available**

End-to-end processing of M/EEG data with BIDS, HED, and EEGLAB **preprint** *Dung Truong*, *Kay Robbins*, *Arnaud Delorme*, and *Scott Makeig* Supporting materials and data are available at **https://osf.io/8brgv/**. Chapter to appear in the book *Methods for analyzing large neuroimaging datasets* edited by Robert Whelan and Herve Lemaitre.

January 3, 2023: **Project page live**

New HED organization homepage goes live at **https://www.hedtags.org**.

6.3 How do you use HED?

HED (Hierarchical Event Descriptors) annotations provide an essential link between experimental data and analysis. HED annotations can be used to describe what happened while data was acquired, participant state, experimental control, task parameters, and experimental conditions. HED annotations are most commonly associated with event files, but these annotations can also be applied to other types of tabular data.

This guide organizes HED resources based on how you might use HED:

- ***As an experimenter***
- ***As a data annotator***
- ***As a data analyst***
- ***As a tool developer***
- ***As a schema builder***

6.3.1 As an experimenter

> … doing experiments and acquiring data:

The lynch-pin of scientific inquiry is the planning and running of experiments to test hypotheses and study behavior. The focus of the discussion here is not explicitly on how an experiment should be designed, but rather on how data should be recorded and transformed to maximize its downstream usability.

Here are some topics of interest to experimenters:

- ***Planning and running an experiment***
- ***Post processing the data***

The *Actionable event annotation and analysis in fMRI: A practical guide to event handling* preprint, which can be found at **https://osf.io/93km8/**, provides concrete guidance and discussion of pitfalls in transforming experimental logs into usable event data. The site includes sample data to use in running the examples.

6.3.1.1 Planning and running an experiment

Most laboratory experiments use neuroimaging equipment and peripheral devices in combination with experiment control software to acquire the experimental data. This section describes some HED tools that may be of use during the log-to-data extraction process. Key questions are:

- What should go into an experimental log?
- How should information about the experimental design and temporal structure be included?
- How will the log data be synchronized with other data?

We assume that event information is primarily contained in experimental logs, whose log entries contain a timestamp, a code, and possibly other information. We assume that this information can be extracted in tabular format. The key point here is:

> Data that isn't recorded is lost forever!

With that caveat in mind, most researchers will run a pilot before the actual experiment to detect issues that might reduce the effectiveness or correctness of the experiment. HED file remodeling tools can help smooth the transition from acquisition to data, both in the pilot and the experiment itself.

Event acquisition

In a traditional neuroimaging experiment that is organized by trial, it may be easy to focus exclusively on marking the experimental stimuli, but the incidental sensory presentations can also be important, particularly for analyses that use regression techniques. Examples of incidental sensory presentations include cues, instructions, feedback, and experimental control events that are visible to the participant.

Participant responses should also be marked in the timeline, even though this may require synchronization of presentation with the acquisition of the participant's response indicators. Downstream analysis may include time-locking to the actual response point to study neural correlates of the motor reaction. A common approach for including participant's response is to identify the closing of a switch on a push-button, marking the end of the participant's response. More sophisticated instrumentation might include detection of initiation and termination of muscle movement using EMG (electromyography) sensors.

Another issue which should be addressed in the pilot is how experimental control information will be embedded in the data. Will there be embedded markers for trial or block beginnings? How will information about experimental conditions be embedded? Often a condition will be counterbalanced within a run and embedding markers that identify the current conditions in the log can facilitate the use of tools in post-processing and assure that the conditions are correctly marked.

Logs to event files

Although the HED tools do not yet directly support any particular experimental presentation/control software packages, the HED *File remodeling tools* can be useful in working with logged data.

Assuming that you can put the information from your experimental log into a tabular form such as:

A sample log file in tabular form.

onset	code	description
0.3423	4332	Presentation of a fixation cross for 0.25 seconds.
0.5923	4333	Presentation of a face image for 0.5 seconds.
1.7000	4332	Presentation of a fixation cross for 0.25 seconds.
…	…	…

The *summarize column values* operation in the HED *file remodeling tools* compiles detailed summaries of the contents of tabular files. Use the following remodeling file and your tabular log file as input to the HED online event remodeling tools to quickly get an overview of its contents.

A sample JSON file with the command to get a summary of the column values in a file.

```
[{
   "operation": "summarize_column_values",
   "description": "Summarize the column values in my log.",
   "parameters": {
       "summary_name": "Log_summary",
       "summary_filename": "Log_summary",
       "skip_columns": ["onset"],
       "value_columns": ["description"]
   }
}]
```

6.3.1.2 Post-processing the event data

The information that first comes off the experimental logs is usually not directly usable for sharing and analysis. A number of HED *File remodeling tools* tools might be helpful for restructuring your first pass at the event files.

The *remap columns* transformation is particularly useful during the initial processing of tabular log information as exemplified by the following example

A sample JSON remodel to create *duration* and *event_type* columns from *code*.

```
[{
    "operation": "remap_columns",
    "description": "Expand the code column.",
    "parameters": {
        "source_columns": ["code"],
        "destination_columns": ["duration", "event_type"],
        "map_list": [["4332", 0.25, "show_cross"],
```

(continues on next page)

(continued from previous page)

```
                    ["4333", 0.50, "show_face"]],
        "ignore_missing": true
    }
}]
```

The result of applying the above transformation to the ***sample tabular log*** file is shown in the following table:

Result of applying remap_columns to the sample tabular log file.

onset	code	description	duration	event_type
0.3423	4332	Presentation of a fixation cross for 0.25 seconds.	0.25	show_cross
0.5923	4333	Presentation of a face image for 0.5 seconds.	0.50	show_face
1.7000	4332	Presentation of a fixation cross for 0.25 seconds.	0.25	show_cross
…	…	…	…	…

The remapping transformation retains all the columns. At this point you can delete and/or reorder columns using other remodeling commands, since BIDS requires that the first two columns in all events files be *onset* and *duration*, respectively. The remodeling JSON file can be expanded to include these transformations as well.

6.3.2 As a data annotator

… organizing data and tagging events:

The move towards open, reproducible science, both in the scientific community and by funding agencies, makes data sharing a requirement. An added benefit, is that data used by others is likely to garner increased recognition and additional citations. This section emphasizes the importance of complete and accurate metadata to enable analysis.

Here are some topics of interest to data annotators:

- ***Standardizing the format***
  - ***Learning about BIDS***
  - ***Learning about HED***
  - ***Integrating HED in BIDS***
- ***Adding HED annotations***
  - ***Viewing available tags***
  - ***Basic annotation strategies***
  - ***More advanced annotations***
- ***Checking correctness***
  - ***Validating HED annotations***
  - ***Checking for consistency***

6.3.2.1 Standardizing the format

An important aspect of data-sharing is putting your data into a standardized format so that tools can read and manipulate the data without the need for special-purpose reformatting code.

BIDS (Brain Imaging data Structure) is a widely used data organization standard for neuroimaging data. HED is well-integrated into the BIDS standard.

Learning about BIDS

- If you are unfamiliar with BIDS, we recommend the **BIDS Start Kit**.
- **Folders and Files** gives an overview of how files in a BIDS dataset are organized.
- The **Annotating a BIDS dataset** tutorial gives an overview of how to get the appropriate metadata into a BIDS dataset.
- See the **BIDS specification** for detailed information on the rules for BIDS. Of special interest to HED annotators are the sections on **Task events** and the **Hierarchical Event Descriptors** appendix.
- There are a variety of tools available to convert to and from BIDS format as summarized in **Software currently supporting BIDS**.

Learning about HED

- The **HED introduction** gives a basic overview of HED's history and goals.
- The **"Capturing the nature of events…"** paper works through a practical example of using HED annotations and suggests best practices for annotation.
- See the **HED specification** for detailed information on the rules for HED. Of special interest to HED users are **Chapter 4: Basic annotation** and **Chapter 5: Advanced annotation**. These chapters explain the different types of HED annotations and the rules for using them.

Integrating HED in BIDS

There are two strategies for incorporating HED annotations in a BIDS dataset:

Method 1: Use a JSON (sidecar) file to hold the annotations.

Method 2: Annotate each line in each event file using the **HED** column.

Method 1 is the typical way that HED annotations are incorporated into a BIDS dataset. The **HED online tools** allow you to easily generate a template JSON sidecar to fill in. The **BIDS annotation quickstart** walks through this process step-by-step.

Method 2 is usually used for instrument-generated annotations or for manual processing (such as users marking bad sections of the data or special features). In both cases the annotations are usually created using special-purpose tools.

When using HED you must provide a HED schema version indicating the HED vocabulary you are using. In BIDS, the schema versions are specified in `dataset_description.json`, a required JSON file that must be placed in the root directory of the dataset. See **HED schema versions** in the BIDS specification for examples.

6.3.2.2 Adding HED annotations

This section discusses the strategy for adding annotations in a BIDS dataset using sidecars. The discussion assumes that you have a JSON sidecar template file ready to annotate. See **BIDS annotation quickstart** for a walk-through of this process.

Viewing available tags

- The HED vocabulary is hierarchically organized as shown in this **expandable view** of the HED standard vocabulary.
- **Schema viewers** gives links to different versions of the HED standard HED vocabularies as well as library vocabularies.

Basic annotation strategies

HED annotations come in variety of levels and complexity. If your HED annotations are in a JSON sidecar, it is easy to start simple and incrementally improve your annotations just by editing the JSON sidecar.

- The **HED annotation quickstart** provides a recipe for creating a simple HED annotation.

A key part of the annotation is to include a good description of each type event. One way to do this is to include a *Description/* tag with a text value as part of each annotation. A good description helps to clarify the information that you want to convey in the tags.

- ***Viewing available tags*** gives options for viewing tags to select.
- **CTAGGER** is a standalone tagging assistant with a user-friendly GUI to ease the tagging process.

More advanced annotations

HED supports a number of advanced annotation concepts which are necessary for a complete description of the experiment.

- **HED definitions**: allow users to define complex concepts. See **HED definitions** for an overview and syntax.
- **Temporal scope**: annotate event processes that extend over time and provide a context for events. Expression of temporal scope is enabled by *Temporal-marker* tags: *Onset*, *Offset*, and *Duration* together with the *Definition* tag. See **Temporal scope** for the rules and usage.
- **Conditions and experimental design**: HED allows users to express annotate experiment design, as well as other information such as task, and the experiment's temporal organization. See **HED conditions and design matrices**.

The **Advanced annotation**) chapter of the HED specification explains the rules for using these more advanced concepts.

6.3.2.3 Checking correctness

Checking for errors is an ongoing and iterative process. It is much easier to build more complex annotations on a foundation of valid annotations. Thus, as you are adding HED annotations, you should frequently revalidate.

Validating HED annotations

- The **HED validation guide** describes the different types of validators available.
- The **HED errors** documentation lists the different types of HED errors and their potential causes.
- The JSON sidecar, which usually contains most of the HED annotations, can be easily validated using the **HED online tools**.
- You should validate the HED annotations separately using the online tools or the HED Python tools before doing a full BIDS validation, as this will make the validation process much simpler.

Checking for consistency

Several HED summary tools allow you to check consistency. The ***Understanding the data*** tutorial in the next section describes some tools that are available to help check the contents of the events files for surprises.

The summary tools are a start, but there are also experiment-specific aspects which ideally should be checked. Bad trial identification is a typical example of experiment-specific checking.

Example of experiment-specific checking.

Suppose each trial in an experiment should consist of a sequence:

> **stimulus–>key-press–>feedback**

You can expect that there will be situations in which participants forget to press the key, press the wrong key, press the key multiple times, or press the key both before and after the feedback.

Ideally, a data annotator would provide information in the event file marking unusual things such as these bad trials, since it is easy for downstream users to improperly handle these situations, reducing the accuracy of analysis.

At this time, your only option is to do manual checks or write custom code to detect these types of experiment-specific inconsistencies. However, work is underway to include some standard types of checks in the HED ***File remodeling tools*** in future releases.

You may also want to reorganize the event files using the remodeling tools. See the ***Remap columns*** a discussion above and links to examples of how to reorganize the information in the columns of the event files.

6.3.3 As a data analyst

> . . . applying HED tools to answer scientific questions:

Whether you are analyzing your own data or using shared data produced by others to answer a scientific question, fully understanding the data and its limitations is essential for accurate and reproducible analysis. This section discusses how HED annotations and tools can be used for effective analysis.

Here are some topics of interest to data analysts:

- ***Understanding the data***
- ***Preparing the data***

- *Analyzing the data*
  - *Factors vectors and selection*
  - *HED analysis in EEGLAB*

6.3.3.1 Understanding the data

Sadly, most currently shared data is under-annotated and may require considerable work and possibly contact with the data authors for correct use and interpretation.

You can get a preliminary sense about what is actually in the data by downloading a single event file (e.g., a BIDS `_events.tsv`) and its associated JSON sidecar (e.g., a BIDS `_events.json`) and creating HED remodeling tool summaries using the *HED online tools for debugging*. Summaries of particular use for analysts include:

- The *column value summary* compiles a summary of the values in the various columns of the event files in the dataset. This summary does not require any HED information.
- The *HED tag summary* creates a summary of the HED tags used to annotate the data.
- The *experimental design summary* gives a summary of the condition variables or other structural tags relating to experimental design, task, or temporal layout of the experiment.

While HED tag summary and the experimental design summaries require that the dataset have HED annotations, these summaries do not rely on the experiment-specific event-coding used in each experiment and can be used to compare information for different datasets.

The *File remodeling quickstart* tutorial gives an overview of the remodeling tools and how to use them. More detailed information can be found in *File remodeling tools*.

The *Online tools for debugging* shows how to use remodeling tools to obtain these summaries without writing any code.

The *HED conditions and design matrices* guide explains how information structure information is encoded in HED and how to interpret the summaries of this information.

6.3.3.2 Preparing the data

In deciding on an analysis, you may discover that the information in the event files is not organized in a way that would support your analyses.

6.3.3.3 Analyzing the data

The power of HED is two-fold – its flexibility and its generality in specifying criteria. Flexibility allows users to specify quite complex criteria without having to write additional code, while generality allows comparison of criteria across different experiments.

The factor generation as described in the next section relies on the HED *File remodeling tools*. See *File remodeling tools*.

Factor vectors and selection

The most common analysis application is to select events satisfying a particular criteria, and compare some measure on signals containing these events with a control. Depending on the modality, these might be different.

HED annotations facilitate the selection. This selection can be described in terms of factor vectors. A **factor vector** for an event file has the same number of rows as the event file (each row corresponding to an event marker). Factor vectors contain 1's for rows in which a specified criterion is satisfied and 0's otherwise.

- The ***factor column operation*** creates factor vectors based on the unique values in specified columns. This factor operation does not require any HED information.
- The ***factor HED tags*** creates factor vectors based on a HED tag query. The ***HED search guide*** explains the HED query structure and available search options.
- The ***factor HED type*** creates factors based on a HED tag representing structural information about the data such as *Condition-variable* (for experimental design and experimental conditions) or *Task*.

HED analysis in EEGLAB

EEGLAB, the interactive MATLAB toolbox for EEG/MEG analysis, supports HED through the ***EEGLAB HEDTools plugin***.

The *End-to-end processing of EEG with HED and EEGLAB* preprint, which can be found at **https://osf.io/8brgv/**, works through the entire analysis process, including porting the analysis to high performance computing platforms. The site includes sample data to use in running the examples.

HED support in other tools

Work is underway to integrate HED support in other analysis packages. If you are interested in helping in this effort please email hed.maintainers@gmail.com.

6.3.4 As a tool developer

… helping expand the growing HED tool base:

The power of HED is its ability to capture important details of the experiment design and events in a form that is both human-understandable and directly usable in processing programs. The HED ecosystem relies on tools that read, understand, and incorporate HED as part of analysis. This section describes how, as a tool developer, you can contribute to this growing ecosystem to support HED for processing and analysis.

Here are some topics of interest to developers:

- ***Integrating with existing tools***
- ***The HED code base***
  - ***The HED Python code base***
  - ***The HED JavaScript code base***
  - ***The HED MATLAB code base***
  - ***Web tools and REST services***
- ***Future development plans***

6.3.4.1 Integration with existing tools

The GitHub repositories and other resources associated with these projects are described in this section. The HED project page is **https://hedtags.org**. The documentation and examples are housed in the **hed-examples** GitHub repository.

Contributions are welcome in any area (e.g., code, examples, documentation, ideas, issues). Use the **issues** mechanism of the most appropriate HED standard repository to ask questions or to describe your ideas and how you would like to contribute. Alternatively, you can email hed.maintainers@gmail.com.

6.3.4.2 The HED code base

The **HED standard organization** has several code projects and distinct tool bases in Python, MATLAB, and JavaScript. All HED efforts are open source.

The HED python code base

The Python HED tools contain the core technology for HED including code for validation, analysis, and schema development. The code for HEDTools is in the **hed-python** GitHub repository.

The latest stable release is available as **hedtools** on PyPI and can be installed using the regular `pip` install mechanism.

The `develop` branch of **hed-python** contains the latest versions of the tools and can be installed from GitHub using:

```
pip install git+https://github.com/hed-standard/hed-python/@develop
```

The HED JavaScript code base

GitHub repository. The JavaScript tools focus on HED validation and its main client is the **Bids validator**. The code for this project is in the **hed-javascript**

The latest stable release is available as the **hed-validator** on npm.

The HED MATLAB code base

The MATLAB HED tools project focuses primarily on analysis using HED, although there is substantial support for annotation as well.

The **HEDTools plugin** is available for installation through **EEGLAB**. The**EEGLAB plug-in integration** tutorial explains the installation and integration of HED tools in the EEGLAB environment. Although this toolset focuses on analysis, it also includes extensive tools for importing and annotating HED data through the **CTagger** GUI.

CTagger is a GUI for HED annotation and validation. CTagger can be run as a standalone program, but is also integrated and callable from MATLAB via an **EEGLAB plug-in**. See **CTAGGER GUI tagging tool** tutorial for more information on installation and use. The project source code is located in the **CTagger** GitHub repository.

HED services in MATLAB explains how the **HED online services** can be called programmatically in MATLAB. The HED services are deployed online through a docker container as described in *Web tools and rest services*.

Python HEDTools in MATLAB explains how to install and call various Python tools from MATLAB.

Web tools and REST services

The HED online tools are available at **https://hedtools.ucsd.edu/hed**.

A development version of the online tools is available at **https://hedtools.ucsd.edu/hed_dev**.

These servers not only provide a GUI interface to the tools that is useful for debugging or for a quick analysis, but they also provide REST services for various HED tools as described in ***HED RESTful services***.

The project source code is located in the **hed-web** GitHub repository.

6.3.4.3 Future development plans

We are always looking for people with suggestions or new ideas to join our community. In the short term we have the following development goals:

- Finish integration of search for epoching and its documentation in **fieldtrip**.
- Integrate searching, summary, and epoching into **MNE-Python**.
- Integrate search and summary into the **Nemar** and **EEGNET** platforms.

Longer term we hope to develop more sophisticated analysis methods based on HED and to better integrate presentation and experimental control software with the annotation process.

We are also tackling the problem of how to effectively capture event relationships to facilitate more complex and sophisticated automated analysis.

6.3.5 As a schema builder

… extending HED vocabulary in new directions:

HED annotations consist of comma-separated terms drawn from a hierarchically structured vocabulary called a HED schema. The **HED standard schema** contains basic terms that are common across most human neuroimaging, behavioral, and physiological experiments.

The HED ecosystem also includes **HED library schemas** to expand the HED vocabulary in a scalable manner to support more specialized data.

Here are some topics of interest to schema developers:

- ***Viewing available schemas***
- ***Improving an existing schema***
- ***Creating a new library schema***
- ***Private vocabularies and extensions***

The SCORE library for clinical EEG annotations has been released. Other schema libraries are under development include a movie annotation library and a language annotation library, but these have not yet reached the stage that they are available for community comment.

If you are interested in participating in the development of any ongoing library development efforts, please email hed.maintainers@gmail.com.

6.3.5.1 Viewing available schemas

The first step in using or improving the HED vocabularies is to explore what is there using the **HED vocabulary viewer** for the HED standard schema.

The SCORE library for clinical annotation of EEG can be viewed using the **HED vocabulary viewer** for score.

6.3.5.2 Improving an existing schema

If you see a need for additional terms in an existing schema, post an issue to schema to **hed-schemas/issues** on GitHub with the following information:

Proposing a new tag in an existing HED schema.

Be sure to include the following when posting an issue to add a schema term.

- The name of the schema (standard or library-name).
- The proposed name of the term or the name of term to be modified.
- A brief and informative text description of its meaning.
- A suggestion for where term should be placed in the schema if new.
- An explanation of why this term is needed and how it might be used.

Proposals for modifications to existing terms should include similar information.

The posting of an issue will start the discussion going. A HED schema term must stand on its own and must not exist elsewhere in the schema. When thinking about where a term should be located within the schema hierarchy, also remember that every term satisfies the **is-a** relationship with any of its schema parents.

Besides adding new terms, you might suggest improvements to an existing term's description or a modification of its attributes. You might also suggest the need for modifications or additions to the schema attributes, value classes, or unit classes.

All suggested changes or errors should be reported using the same mechanism as proposing new terms through the **hed-schemas/issues** mechanism on GitHub.

6.3.5.3 Creating a new library schema

If you are interested in developing a library schema in a new area, you should post an issue on the **hed-schemas/issues** GitHub repository. Your post should start with a brief description of the proposed library and its applications.

Starting the process of developing a new HED schema library.

Be sure to include the following for your initial post proposing creation of a new library.

- A proposed name for the HED library schema.
- A brief description of the library's purpose and contents.
- GitHub handles for potential collaborators.

You should also read the **HED schema development guide** to get an overview of the development process.

Note: You must have a GitHub account in order to work on the development of a new schema as all development processes for HED use the GitHub Pull Request mechanism for development and community comment.

6.3.5.4 Private vocabularies and extensions

Although you can create a private HED vocabulary for your own use, many HED tools assume that only standardized schemas available on the **hed-schemas** GitHub repository will be used. These tools fetch or internally cache the most recent versions of the HED schemas, and users need only specify the HED schema versions during validation and analysis.

The decision to only support standardized schemas was after serious deliberation by the HED Working Group based on the observation that the ability of HED to enable standardized dataset summaries and comparisons would be compromised by allowing unvetted, private vocabularies.

6.4 BIDS annotation quickstart

This tutorial provides a step-by-step guide to creating a JSON sidecar containing the annotations needed to document your BIDs dataset events. See ***HED annotation quickstart*** for guidelines on what annotations to choose.

We assume that your dataset is already in the BIDS **BIDS Brain Imaging Data Structure** format and focus on the mechanics of event annotation in BIDS using HED.

General strategy for machine-actionable annotation using HED.

The goal is to construct a single `events.json` sidecar file with all the annotations needed for users to understand and analyze your data.

You will put the finished annotation file at the top level of your dataset.

The approach that we will use is to create a template file from an `events.tsv` file in your BIDS dataset using the online tools available at **hedtools.ucsd.edu/hed**.

You can then edit this JSON file directly using a text editor to insert data descriptions and HED annotations.

You also have the option of converting this JSON template to a spreadsheet template for editing convenience as described below in ***Spreadsheet templates***.

Warning: Although the HED web tools base the template on the information extracted from a single `events.tsv` file, this will be sufficient to produce a good template for most datasets.

For datasets with widely-varying event files, you should use the bids_validate_hed.ipynb Jupyter notebook version rather than the online tools. The Jupyter notebook consolidates information from all of the `events.tsv` files in the dataset to produce a comprehensive JSON sidecar template.

The examples in this tutorial use an **abbreviated version** of the `events.tsv`file from subject 002 run 1 from **ds003645**:Face processing MEEG dataset with HED annotation dataset on OpenNeuro. A reduced version of this dataset **eeg_ds003645s_hed** is also available.

6.4.1 How HED works in BIDS

Before getting into the details of event annotation, we briefly explain how BIDS represents events.

6.4.1.1 BIDS event files

BIDS events are time markers with associated metadata stored in tabular form in `events.tsv` files. Each `events.tsv` file is associated with a particular data recording file by its filename using the BIDS naming scheme and by its location within the BIDS dataset directory tree.

For example, the file `sub_002_task-FacePerception-run-1_events.tsv` gives event markers relative to the EEG data file `sub_002_task-FacePerception-run-1_eeg.set` located in the same directory because the file names match up to the last underbar.

The following is an excerpt of a BIDS events file showing its tabular structure.

A simplified excerpt from a BIDS events file.

onset	duration	sample	event_type	face_type	rep_status	trial	rep_lag	value	stim_file
0.004	n/a	1	setup_right_sym	n/a	n/a	n/a	n/a	3	n/a
24.2098	n/a	6052	show_face	unfamiliar_face	first_show	1	n/a	13	u032.bmp
25.0353	n/a	6259	show_circle	n/a	n/a	1	n/a	0	circle.bmp
25.158	n/a	6290	left_press	n/a	n/a	1	n/a	256	n/a
…									

BIDS requires that all `events.tsv` files have an `onset` column containing the time (in seconds) of the event relative to the start of the data recording to which it is linked.

BIDS also requires a `duration` column giving the duration in seconds of the event associated with this event marker.

BIDS uses `n/a` to designate values that should be ignored.

The BIDS specification also mentions several optional columns, but validation of appropriate use is not done by the BIDS validator.

The exception is the optional `HED` column, which is used for event-specific annotations and verified with the BIDS validator.

Users are also free to add their own columns to any `events.tsv` file. This flexibility in the format of the `events.tsv` files is necessary to accommodate the variety of possible events across the spectrum of BIDS datasets, but it complicates data handling for downstream users, who won't know the meaning of these events.

Luckily, BIDS provides a mechanism for describing events in a machine-understandable, validated format using JSON sidecars and HED (Hierarchical Event Descriptors).

6.4.1.2 JSON event sidecars

The BIDS `events.json` files provide the BIDS mechanism for machine-actionable event processing, meaning that downstream users can analyze the data with appropriate tools without writing a lot of code.

Here is an excerpt from a BIDS `events.json` sidecar that is associated with the above *events.tsv* excerpt.

```
{
    "event_type": {
        "Description": "The main category of the event.",
        "HED": {
            "setup_right_sym": "Experiment-structure, Condition-variable/Right-key-
↪assignment",
            "show_face": "Sensory-event, Experimental-stimulus, Visual-presentation,␣
↪Image, Face",
            "left_press": "Agent-action, Participant-response, (Press, Keyboard-key)",
            "show_circle": "Sensory-event, (White, Circle), (Intended-effect, Cue)"
        },
        "Levels": {
            "setup_right_sym": "Right index finger key press means above average␣
↪symmetry.",
            "show_face": "Display a stimulus face image.",
            "left_press": "Participant presses a key with left index finger.",
            "show_circle": "Display a white circle on black background."
        }
    },
    "face_type": {
        "Description": "Factor indicating type of face image being displayed.",
        "Levels": {
          "famous_face": "A face that should be recognized by the participants.",
          "unfamiliar_face": "A face that should not be recognized by the participants.",
          "scrambled_face": "A scrambled face image generated by taking face image 2D␣
↪FFT."
        },
        "HED": {
            "famous_face": "(Condition-variable/Famous-face, (Image, (Face, Famous)))",
            "unfamiliar_face": "(Condition-variable/Unfamiliar-face, (Image, (Face,␣
↪Unfamiliar)))",
            "scrambled_face": "(Condition-variable/Scrambled-face,  (Image, (Face,␣
↪Disordered)))"
        }
    },
    "stim_file": {
        "Description": "Filename of the presented stimulus image.",
        "HED": "(Image, Pathname/#)"
    }
}
```

The JSON sidecar is a dictionary, where the keys correspond to column names.

In the above example, we have provided annotations for the columns `event_type`, `face_type`, and `stim_file`. The values corresponding to these keys are dictionaries of relevant metadata about the corresponding columns.

Several columns in the `events.tsv` file do not have keys in the JSON sidecar (`onset`, `duration`, `sample`, `rep_status`, `trial`, `rep_lag`, `value`) because we have chosen not to provide information about these columns.

The `Description` fields provide information about the general meanings of the corresponding columns.

The `Levels` fields provide information about individual categorical values within a column in a human-readable form.

The `HED` sidecar fields contain descriptive tags from a controlled vocabulary, which can be read and processed by computer algorithms.

At analysis time, tools are available to assemble the HED annotations for each event.

For example the relevant HED tags for the second event in the ***excerpted event file*** are:

> **show_face**: *Sensory-event, Experimental-stimulus, Visual-presentation, Image, Face*
> **unfamiliar_face**: *(Condition-variable/Unfamiliar-face, (Image, (Face, Unfamiliar)))*
> **stim_file**: *(Image, Pathname/#)*

The `stim_file` column has been annotated as a value column rather a categorical column, so the HED tags corresponding to that column are assembled by substituting the actual column value in the `events.tsv` file for the `#` tag placeholder. HED tools can assemble the complete annotation for each event from the event file and its accompanying JSON sidecar.

Final assembled HED tags for second event in the excerpted event file.

Sensory-event, Experimental-stimulus, Visual-presentation, Image, Face,
(Condition-variable/Unfamiliar-face, (Image, (Face, Unfamiliar))),
(Image, Pathname/u032.bmp)

The standardized HED vocabulary allows tools to search for events with common tags across datasets.

We recommend that when at all possible, you place your HED annotations in a single JSON sidecar file located in the root directory of your BIDS dataset.

Do not use the `HED` column in the individual `events.tsv` unless you really need to annotate events individually, because individual event annotation is a lot more work and harder to maintain.

The next section guides you through the creation of a JSON sidecar for event annotation using convenient online tools.

The ***Basic HED Annotation*** tutorial walks you through the process of selecting HED tags for annotation.

6.4.2 Create a JSON template

As described in the previous section, users provide metadata about events in a JSON sidecar. This tutorial demonstrates how to use online tools to generate a JSON sidecar template by extracting information from one of the `events.tsv` files in your BIDS dataset. Once the skeleton of the JSON sidecar is in place, and you just need to edit in your specific metadata.

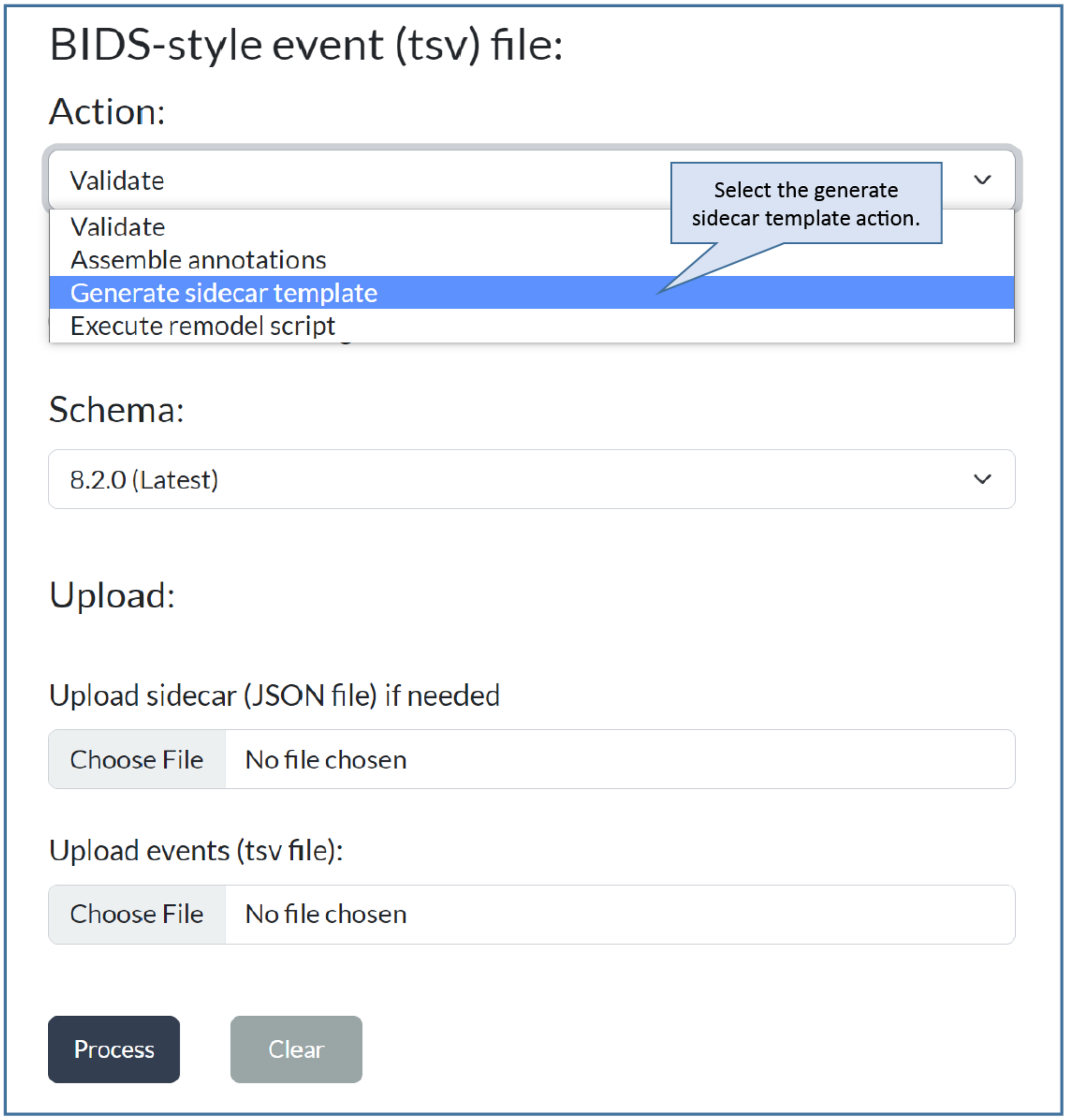
Working from a template is **much easier and faster** than creating a sidecar from scratch. Using the **HED events online tools**, the steps to create a template are:

- ***Step 1: Select generate JSON.***
- ***Step 2: Upload an event file.***
- ***Step 3: Select columns to annotate.***
- ***Step 4: Download the extracted template.***
- ***Step 5: Complete the annotation.***

You can then edit your JSON sidecar directly or convert it to a spreadsheet to fill in the annotations.

6.4.2.1 Step 1: Select generate JSON

Go to the **Events** page of the HED online tools. You will see the following menu:


BIDS-style event (tsv) file:

Action:

Validate

Validate
Assemble annotations
Generate sidecar template
Execute remodel script

Select the generate sidecar template action.

Schema:

8.2.0 (Latest)

Upload:

Upload sidecar (JSON file) if needed

Choose File No file chosen

Upload events (tsv file):

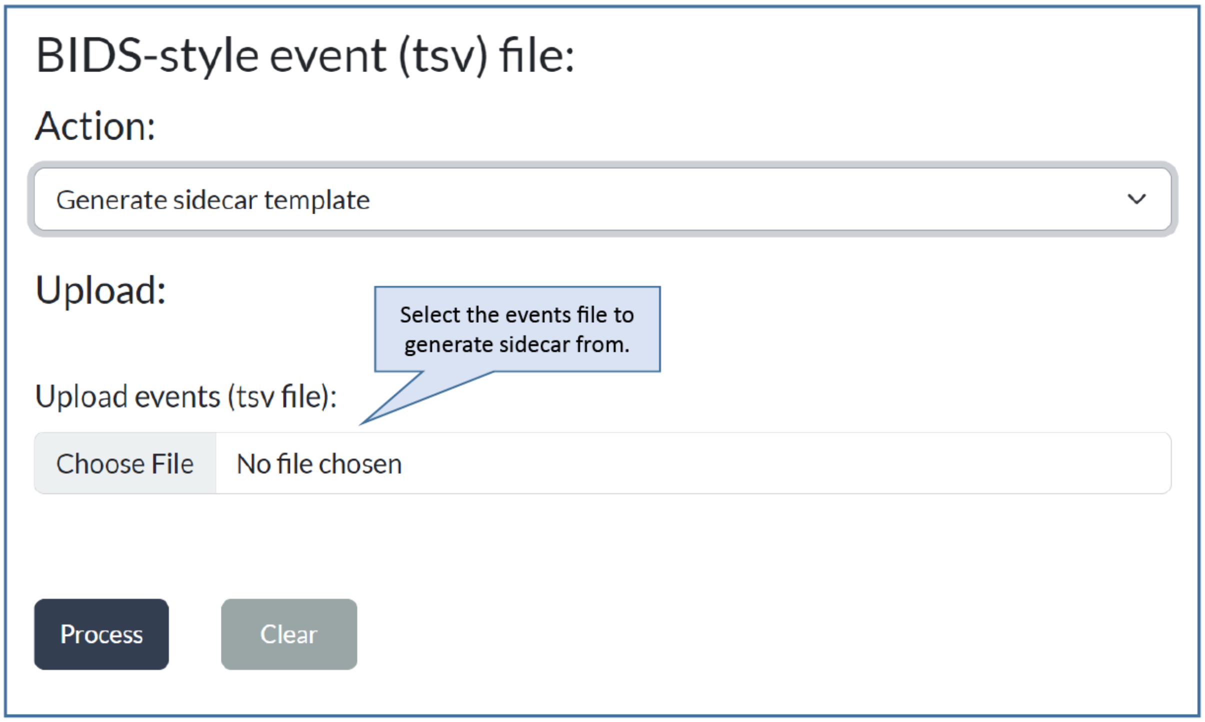
Choose File No file chosen

Process Clear

Select **Generate sidecar template**. The application will adjust to your selection, showing only the information you need to provide.

6.4.2.2 Step 2: Upload an events file.

Use the **Browse** button to choose an `events.tsv` file to upload. When the upload is complete, the local file name of the uploaded events file will be displayed next to the **Browse** button.


BIDS-style event (tsv) file:
Action:
Generate sidecar template
Upload:
Select the events file to generate sidecar from.
Upload events (tsv file):
Choose File
No file chosen
Process
Clear

In this example, we have uploaded **sub-002_task-FacePerception_run-1_events.tsv**. Here is a simplified excerpt from the beginning of this file:

A simplified excerpt from a BIDS event file.

onset	duration	sample	event_type	face_type	rep_status	trial	rep_lag	value	stim_file
0.004	n/a	1.0	setup_right_sym	n/a	n/a	n/a	n/a	3	n/a
24.2098	n/a	6052	show_face	unfamiliar_face	first_show	1	n/a	13	u032.bmp
25.0353	n/a	6259	show_circle	n/a	n/a	1	n/a	0	circle.bmp
25.158	n/a	6290	left_press	n/a	n/a	1	n/a	256	n/a
…									

When the upload is complete, the application will expand to show the columns present in the uploaded `events.tsv` file.

6.4.2.3 Step 3: Select columns to annotate

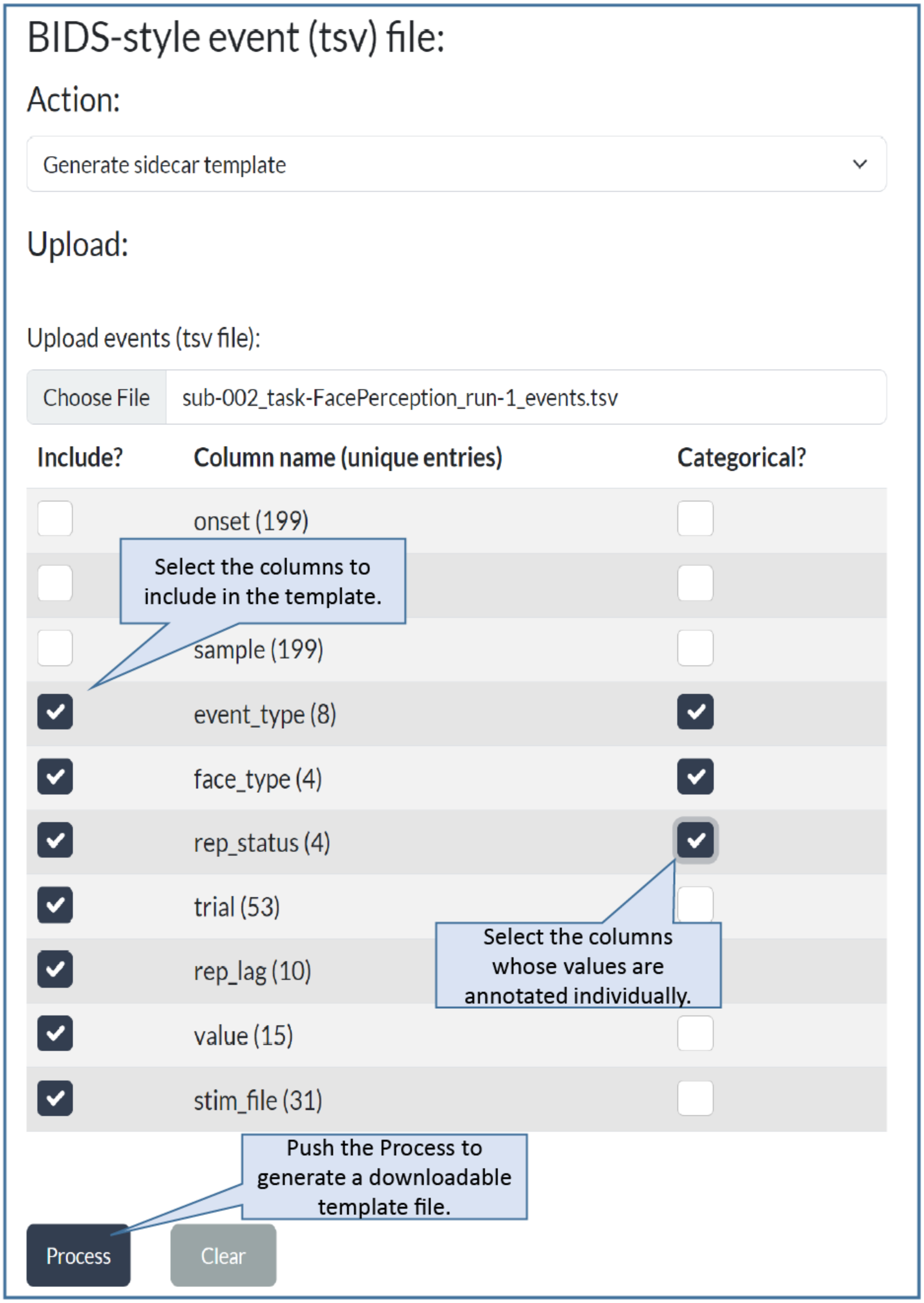
Annotations consist of descriptions of the values in the `events.tsv` file as well as associated HED tags that allow computer tools to directly process these.

You will use the summary information provided about the columns in the `events.tsv` file to decide which columns should be annotated.

The checkboxes on the left indicate which columns should be included in the JSON sidecar annotation template.

The checkboxes on the right indicate which event file columns contain values that you wish to annotate individually. We refer to these columns as the **categorical** columns.

The numbers in parentheses next to the column names give the number of unique values in each column. You will not want to treat columns with a large number of unique values as categorical columns, since you will need to provide an individual annotation for each value in such a categorical column.



BIDS-style event (tsv) file:

Action:

Generate sidecar template

Upload:

Upload events (tsv file):

Choose File sub-002_task-FacePerception_run-1_events.tsv

Include?	Column name (unique entries)	Categorical?
☐	onset (199)	☐
☐		☐
☐	sample (199)	☐
✓	event_type (8)	✓
✓	face_type (4)	✓
✓	rep_status (4)	✓
✓	trial (53)	☐
✓	rep_lag (10)	
✓	value (15)	☐
✓	stim_file (31)	☐

Select the columns to include in the template.

Select the columns whose values are annotated individually.

Push the Process to generate a downloadable template file.

In the example, we have selected 7 columns to annotate. We omitted the `onset`, `duration`, and `sample` columns, since these columns have standardized meanings. The `duration` column has only 1 unique value because this particular dataset has `n/a` for all entries.

We have selected the `event_type`, `face_type`, and `rep_status` columns as categorical columns, meaning that we will annotate each unique value in these columns in a separate annotation. The `event_type`, `face_type`, and `rep_status` have a total of 16 unique values.

In addition, we have elected to annotate `trial`, `rep_lag`, `value`, and `stim_file` by describing these columns as a whole, resulting in 4 additional annotations.

In all, we will have to provide a total of 8 + 4 + 4 + 1 + 1 + 1 + 1 = 20 HED annotations based on the selections we have made.

6.4.2.4 Step 4: Download the template.

After you press the **Process** button, the online tools produce a JSON template file for you download. Save the file, and you are ready to begin the actual annotation. You can edit the JSON sidecar using a text editor or other appropriate tool.

The **sub-002_task-FacePerception_run-1_events.tsv** file generates this **JSON sidecar template**. The following is a simplified excerpt of this template, which we will use to illustrate the rest of the annotation process.

JSON sidecar generated template.

```
{
    "event_type": {
        "Description": "Description for event_type",
        "HED": {
            "setup_right_sym": "(Label/event_type, Label/setup_right_sym)",
            "left_press": "(Label/event_type, Label/left_press)",
            "show_face": "(Label/event_type, Label/show_face)",
            "show_circle": "(Label/event_type, Label/show_circle)"
        },
        "Levels": {
            "setup_right_sym": "Description for setup_right_sym of event_type",
            "left_press": "Description for left_press of event_type",
            "show_face": "Description for show_face of event_type",
            "show_circle": "Description for show_circle of event_type"
        }
    },
    "stim_file": {
        "Description": "Description for stim_file",
        "HED": "(Label/stim_file, Label/#)"
    }
}
```

Notice the difference in structure between annotations for columns that are designated as categorical columns (such as `event_type`) and columns that are designated as value columns (such as `stim_file`). The HED annotations for the non-categorical value columns must contain a # so that the individual column values can be substituted for the # placeholder when the annotation is assembled.

6.4.2.5 Step 5: Complete the annotation.

Once you have a JSON sidecar template, you should edit in your event annotations. The following is an edited version of the *simplified template excerpt* containing a minimal set of HED annotations.

JSON sidecar with completed annotation.

```
{
    "event_type": {
        "Description": "The main category of the event.",
        "HED": {
            "setup_right_sym": "Experiment-structure, Condition-variable/Right-key-
↪assignment",
            "left_press": "Agent-action, Participant-response, (Press, Keyboard-key)",
            "show_face": "Sensory-event, Experimental-stimulus, Visual-presentation,␣
↪Image, Face",
            "show_circle": "Sensory-event, (White, Circle), (Intended-effect, Cue)"
        },
        "Levels": {
            "setup_right_sym": "Right index finger key press means above average␣
↪symmetry.",
            "left_press": "Participant presses a key with left index finger.",
            "show_face": "Display a stimulus face image.",
            "show_circle": "Display a white circle on black background."
        }
    },
    "stim_file": {
        "Description": "Filename of the presented stimulus image.",
        "HED": "(Image, Pathname/#)"
    }
}
```

If you feel comfortable working with JSON files you can edit the HED annotations and descriptions directly in the JSON file.

The HED annotations in the examples are minimal to simplify the explanations. See ***Basic HED Annotation*** for guidelines on how to select HED tags.

Once you have finished, you should validate your JSON file to make sure that your annotations are correct. See the ***HED validation guide*** for detailed guidance. When you are satisfied with your valid JSON sidecar, simply upload it to the root directory of your BIDS dataset, and you are done.

If you would rather work with spreadsheets when doing your annotations, you can extract a spreadsheet from the JSON sidecar to edit and merge back after you are finished. This process is described in the next section, which you can skip if you are going to edit the JSON directly.

6.4.3 Spreadsheet templates

Many people find working with a spreadsheet of annotations easier than direct editing a JSON events sidecar file. The HED online tools provide an easy method for converting between a JSON sidecar and a spreadsheet representation.

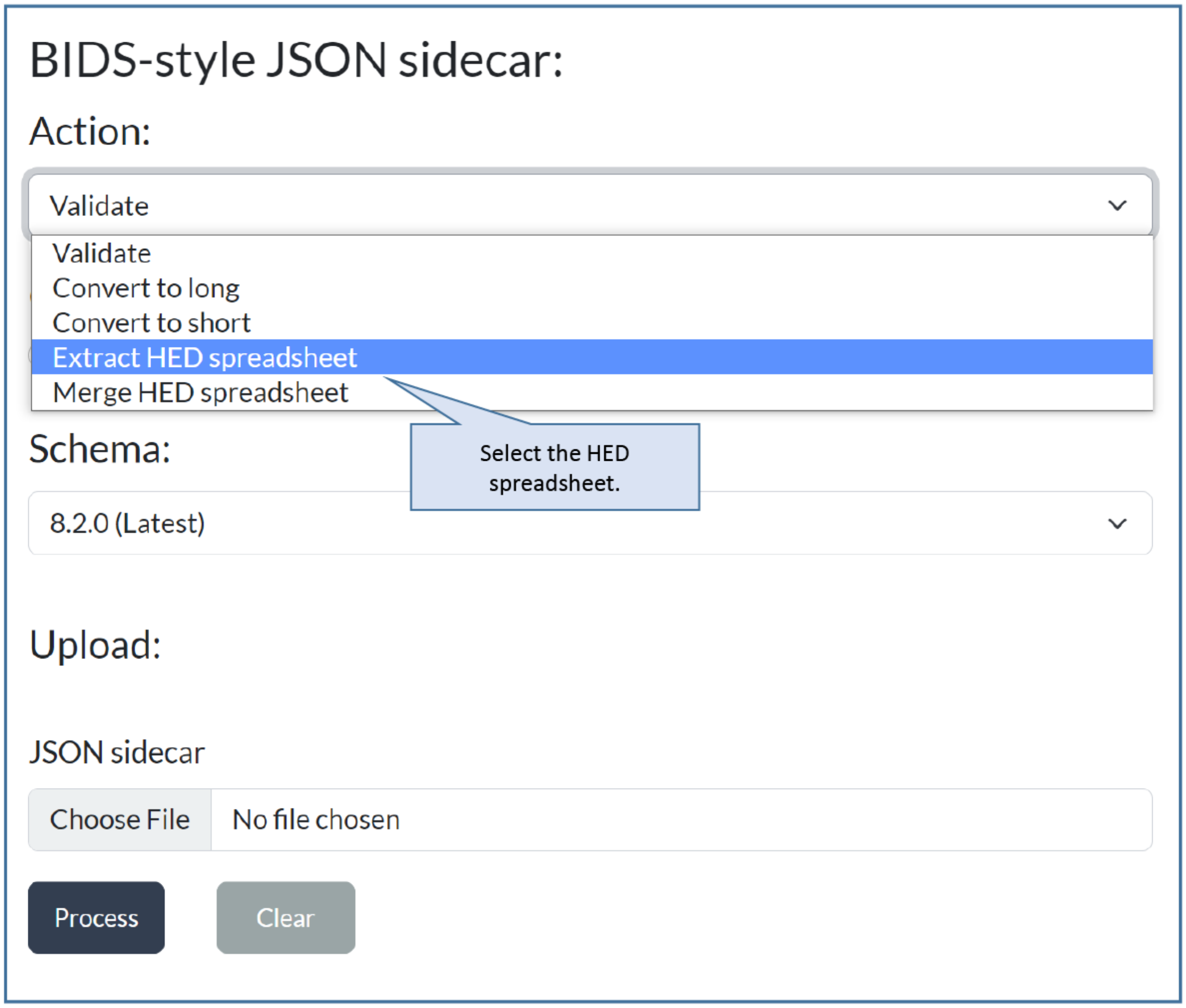
You can convert the JSON events sidecar file into a spreadsheet for easier editing and then convert back to a JSON file afterwards. This tutorial assumes that you already have a JSON events sidecar or have *extracted a JSON sidecar template*.

Using the **HED sidecar online tools**, the steps to create a template are:

- *Step 1: Select extract HED spreadsheet.*
- *Step 2: Upload a sidecar and extract.*
- *Step 3: Edit the spreadsheet.*
- *Step 4: Merge the spreadsheet.*

6.4.3.1 Step 1: Select extract HED spreadsheet

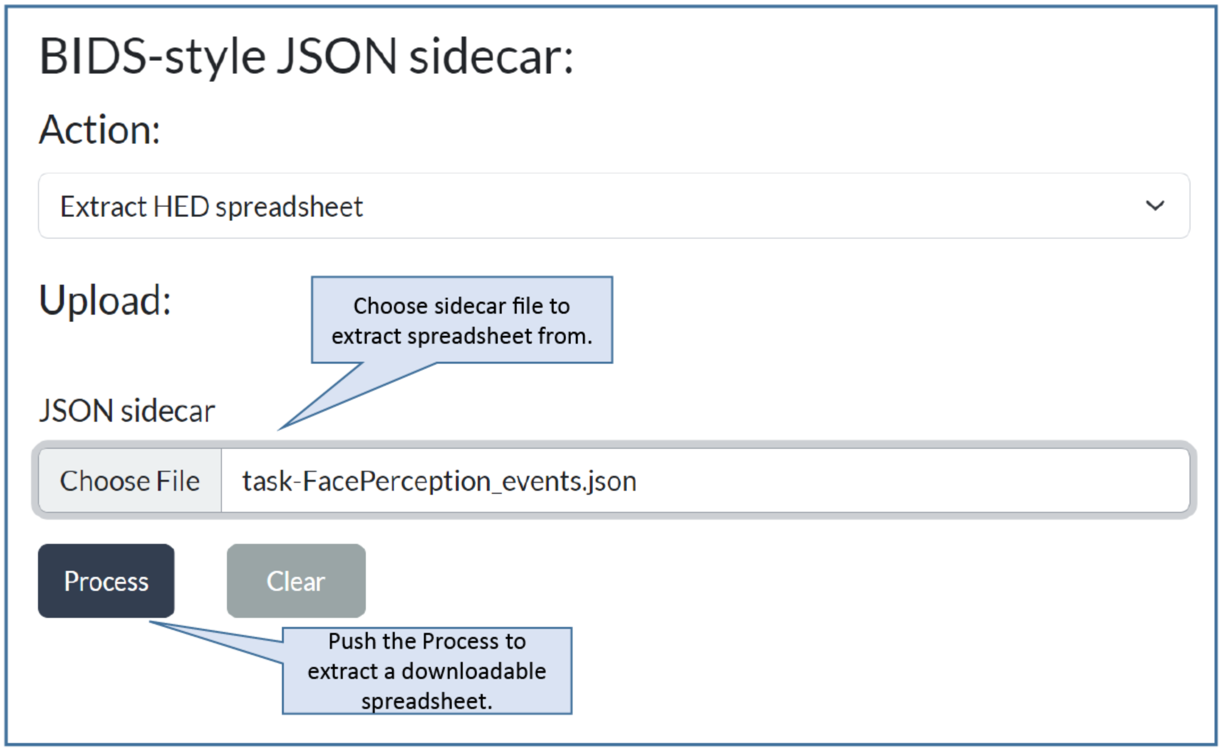
Go to the **Sidecar** page of the HED online tools. You will see the following menu:



Select **Extract HED spreadsheet**. The application will adjust to your selection, showing only the information you need to provide.

6.4.3.2 Step 2: Upload a sidecar and extract.

Use the **Browse** button to choose an `events.json` file to upload. When the upload is complete, the local file name of the uploaded events file will be displayed next to the **Browse** button.


BIDS-style JSON sidecar:
Action:
Extract HED spreadsheet
Upload:
Choose sidecar file to extract spreadsheet from.
JSON sidecar
Choose File
task-FacePerception_events.json
Process
Clear
Push the Process to extract a downloadable spreadsheet.

Pressing the **Process** button causes the application to generate a downloadable tab-separated-value spreadsheet for editing

An excerpt from the **spreadsheet** generated from the **extracted JSON file** is:

HED annotation table extracted from JSON sidecar template.

column_name	column_value	description	HED
event_type	setup_right_sym	Description for setup_right_sym	(*Label/event_type*, *Label/setup_right_sym*)
event_type	show_face	Description for show_face	(*Label/event_type*, *Label/show_face*)
event_type	left_press	Description for left_press	(*Label/event_type*, *Label/left_press*)
event_type	show_circle	Description for show_circle	(*Label/event_type*, Label/show_circle)
stim_file	n/a	Description for stim_file	*Label/#*

The spreadsheet has 4 columns: the **column_name** corresponds to the column name in the `events.tsv` file. The **column_value** corresponds to one of the unique values within that column. The **description** column is used to fill in the corresponding `Levels` value, while the **HED** column is used for the **HED** tags that make your annotation machine-actionable. These tags are from the corresponding `HED` entry in the sidecar.

The last row of the excerpt has `stim_file` as the **column_name**. This column was not selected as a categorical column when the sidecar template was created. The **column_value** for such columns is always `n/a`. The **description** column is used for the `Description` value in the sidecar. The **HED** column tags must include a `#` placeholder in this case. During analysis the column value is substituted for the `#` when the HED annotation is assembled.

6.4.3.3 Step 3: Edit the spreadsheet

After saving the file, you are free to edit it in a text editor or in a tool such as Excel. You may save the edited spreadsheet in either `.tsv` or `.xslx` format.

The following is the extracted spreadsheet corresponding to the *edited JSON sidecar above*.

HED annotation table extracted from JSON sidecar template.

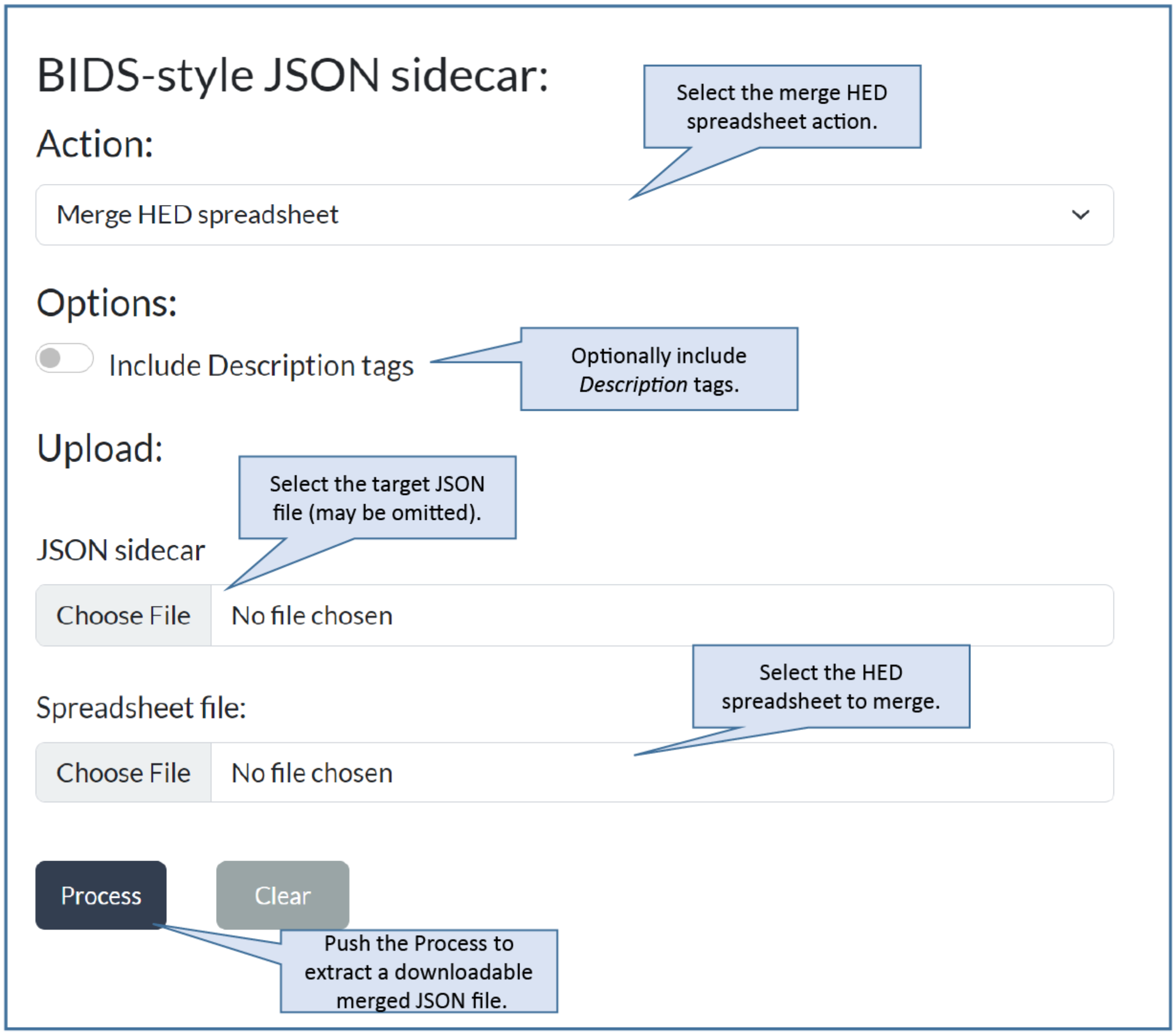
column_name	column_value	description	HED
event_type	setup_right_sym	Right index finger key pressmeans above average symmetry.	*Experiment-structure*,*Condition-variable/Right-key-assignment*
event_type	show_face	Display a stimulus face image.	*Sensory-event*, *Experimental-stimulus*,*Visual-presentation*, *Image*, *Face*
event_type	left_press	Participant presses keywith left index finger.	*Agent-action*, *Participant-response*,*(Press, Keyboard-key)*
event_type	show_circle	Display a white circle on black background.	*Sensory-event*, (*White*, *Circle*),(*Intended-effect*, *Cue*)
stim_file	n/a	Filename of the presented stimulus image.	(*Image*, *Pathname/#*)

If you wish a particular table cell to be ignored, use `n/a` in the cell.

6.4.3.4 Step 4: Merge the spreadsheet

Although editing metadata in a spreadsheet is convenient, BIDS stores all of its metadata in JSON files. If you choose to extract a spreadsheet for editing your annotations, you will need to merge the edited spreadsheet back into a JSON sidecar before including it in your BIDS dataset.

Using the **HED sidecar online tools**, select merge HED spreadsheet as shown below. You may choose an existing edited sidecar, the original template, or an empty sidecar as the JSON target file for the merge.



BIDS-style JSON sidecar:

Action:

Merge HED spreadsheet

Options:

Include Description tags

Upload:

JSON sidecar

Choose File No file chosen

Spreadsheet file:

Choose File No file chosen

Select the merge HED spreadsheet action.
Optionally include *Description* tags.
Select the target JSON file (may be omitted).
Select the HED spreadsheet to merge.
Process
Clear
Push the Process to extract a downloadable merged JSON file.

Pressing the **Process** button causes the application to generate a downloadable version of the merged JSON file.

The merging process replaces the `HED` section of the JSON file for a specified column name and column value with the tags in the corresponding **HED** column of the spreadsheet.

Similarly, merging replaces the `Levels` section of the JSON file for a specified column name and column value with the description in the corresponding **description** column of the spreadsheet. For value columns, the description replaces the value of the `Description` entry corresponding to that column.

Since the BIDS JSON sidecar files may contain other information besides HED annotations, the merging process tries to preserve the sidecar entries that are not directly related to the HED annotations. The merging process also ignores **description** and **HED** spreadsheet entries containing `n/a`.

Notice that there is an option to include *Description* tags when doing the merge. If this box is checked, the contents of the **description** field are prepended with the *Description* tag and appended to the tags.

6.5 HED annotation quickstart

This tutorial takes you through the steps of annotating the events using HED (Hierarchical Event Descriptors). The tutorial focuses on how to make good choices of HED annotations to make your data usable for downstream analysis. The mechanics of putting your selected HED annotations into BIDS (Brain Imaging Data Structure) format is covered in the ***BIDS annotation quickstart*** guide.

- ***What is HED annotation?***
- ***A recipe for simple annotation***

6.5.1 What is HED annotation?

A HED annotation consists of a comma separated list of tags selected from a HED vocabulary or schema. An important reason for using an agreed-upon vocabulary rather than free-form tagging for annotation is to avoid confusion and ambiguity and to promote data-sharing.

The basic terms are organized into trees for easier access and search. The **Expandable HED vocabulary viewer** allows you to explore these terms.

6.5.2 A recipe for simple annotation

In thinking about how to annotate an event, you should always start by selecting a tag from the *Event* subtree to indicate the general event category. Possible choices are: *Sensory-event*, *Agent-action*, *Data-feature*, *Experiment-control*, *Experiment-procedure*, *Experiment-structure*, and *Measurement-event*. See the **Expandable HED vocabulary viewer** to view the available tags.

Most experiments will only have a few types of distinct events. The simplest way to create a minimal HED annotation for your events is:

1. Select one of the 7 tags from the *Event* subtree to designate the general category of the event.
2. Use the following table to select the appropriate supporting tags given that event type.

Standard HED tag selections for minimal annotation.

Event tag	Support tag type	Example tags	Reason
Sensory-event	*Sensory-presentation*	*Visual-presentationAuditory-presentation*	Which sense?
	Task-event-role	*Experimental-stimulusInstructional*	What task role?
	Task-stimulus-role	*CueTarget*	Stimulus purpose?
	Item	*(Face, Image)Siren*	What is presented?
	Sensory-attribute	*Red*	What modifiers are needed?
Agent-action	*Agent-task-role*	*Experiment-participant*	Who is agent?
	Action	*MovePress*	What action is performed?
	Task-action-type	*Appropriate-actionNear-miss*	What task relationship?
	Item	*ArmMouse-button*	What is action target?
Data-feature	*Data-source-type*	*Expert-annotationComputed-feature*	Where did the feature come from?
	Label	*Label/Blinker_BlinkMax*	Tool name?Feature type?
	Data-value	*Percentage/32.5 Time-interval/1.5 s*	Feature value or type?
Experiment-control	*Agent*	*Controller-Agent*	What is the controller?
	Informational	*Label/Stop-recording*	What did the controller do?
Experiment-procedure	*Task-event-role*	*Task-activity*	What procedure?
Experiment-structure	*Organizational-property*	*Time-blockCondition-variable*	What structural property?
Measurement-event	*Data-source-type*	*Instrument-measurementObservation*	Source of the data.
	Label	*Label/Oximeter_O2Level*	Instrument name?Measurement type?
	Data-value	*Percentage/32.5 Time-interval/1.5 s*	What value or type?

As in BIDS, we assume that the event metadata is given in tabular form. Each table row represents the metadata associated with a single data event marker, as shown in the following excerpt of the `events.tsv` file for a simple Go/No-go experiment. The `onset` column gives the time in seconds of the marker relative to the beginning of the associated data file.

Event file from a simple Go/No-go experiment.

onset	duration	event_type	value	stim_file
5.035	n/a	stimulus	animal_target	105064.jpg
5.370	n/a	response	correct_response	n/a
6.837	n/a	stimulus	animal_distractor	38068.jpg
8.651	n/a	stimulus	animal_target	136095.jpg
8.940	n/a	response	correct_response	n/a
10.801	n/a	stimulus	animal_distractor	38014.jpg
12.684	n/a	stimulus	animal_distractor	82063.jpg
12.943	n/a	response	incorrect_response	n/a

In the Go/No-go experiment, the experimental participant is presented with a series of target and distractor animal images. The participant is instructed to lift a finger off a button when a target animal image appears. Since in this experiment, the `value` column has distinct values for all possible unique event types, the `event_type` column is redundant. In this case, we can choose to assign all the annotations to the `value` column as demonstrated in the following example.

Version 1: Assigning all annotations to the value column.

value	Event category	Supporting tags
animal_target	*Sensory-event*	*Visual-presentation*, *Experimental-stimulus*,*Target*, (*Animal*, *Image*)
animal_distractor	*Sensory-event*	*Visual-presentation*, *Experimental-stimulus*,*Non-target*, *Distractor*, (*Animal*, *Image*)
correct_response	*Agent-action*	*Experiment-participant*, (*Lift*, *Finger*), *Correct-action*
incorrect_response	*Agent-action*	*Experiment-participant*, (*Lift*, *Finger*), *Incorrect-action*

The table above shows the event category and the supporting tags as suggested in the ***Standard hed tags for minimal annotation*** table.

A better format for your annotations is the ***4-column spreadsheet format*** described in ***BIDS annotation quickstart***, since there are online tools to convert this format into a JSON sidecar that can be deployed directly in a BIDS dataset.

4-column spreadsheet format for the previous example.

column_name	column_value	description	HED
value	animal_target	An target animal image waspresented on a screen.	*Sensory-event*, *Visual-presentation*,*Experimental-stimulus*,*Target*, (*Animal*, *Image*)
value	animal_distractor	A non-target animal distractorimage was presentedon a screen.	*Sensory-event*, *Visual-presentation*,*Experimental-stimulus*, *Non-target*,*Distractor*, (*Animal*, *Image*)
value	correct_response	Participant correctlylifted finger off button.	*Agent-action*, *Experiment-participant*,(*Lift*, *Finger*), *Correct-action*
value	incorrect_response	Participant lifted finger offthe button but should not have.	*Agent-action*, *Experiment-participant*,(*Lift*, *Finger*), *Incorrect-action*

HED tools assemble the annotations for each event into a single HED tag string. An exactly equivalent version of the previous example splits the HED tag annotation between the `event_type` and `value` columns as shown in the next example.

Version 2: Assigning annotations to multiple event file columns.

column_name	column_value	description	HED
event_type	stimulus	An image of an animalwas presented on acomputer screen.	*Sensory-event*, *Visual-presentation*,*experimental-stimulus*
event_type	response	Participant lifted fingeroff button.	*Agent-action*, *Experiment-participant*,(*Lift*, *Finger*)
value	animal_target	A target animal image.	*Target*, (*Animal*, *Image*)
value	animal_distractor	A non-target animal imagemeant as a distractor.	*Non-target*, *Distractor*,(*Animal*, *Image*)
value	correct_response	The previous stimuluswas a target animal.	*Correct-action*
value	incorrect_response	The previous stimuluswas not a target animal.	*Incorrect-action*
stim_file	n/a	Filename of stimulus image.	(*Image*, *Pathname/#*)

In version 2, the annotations that are common to all stimuli and responses are assigned to `event_type`. We have also included the annotation for the `stim_file` column in the last row of this table.

The assembled annotation for the first event (with onset 5.035) in the ***event file excerpt from go/no-go*** above is:

> *Sensory-event*, *Visual-presentation*, *Experimental-stimulus*, *Target*, (*Animal*, *Image*), (*Image*, *Pathname/105064.jpg*)

Mapping annotations and column information across multiple column values often makes the annotation process simpler, especially when annotations become more complex. Multiple column representation also can make analysis easier, particularly if the columns represent information such as design variables.

See ***BIDS annotation quick start*** for how to create templates to fill in with your annotations using online tools. Once you have completed the annotation and converted it to a sidecar, you simply need to place this sidecar in the root directory of your BIDS dataset.

This quick start demonstrates the most basic HED annotations. HED is capable of much more extensive and expressive annotations as explained in a series of tutorials on this site.

6.6 HED validation guide

6.6.1 What is HED validation?

HED validation is the process of checking the consistency and usage of HED annotations.

You should be sure to validate your data before applying analysis tools. Most HED analysis tools, such as those used for searching, summarizing, or creating design matrices, assume that the dataset and its respective event files have already been validated and do not re-validate during analysis.

This guide explains the types of errors that can occur and various ways that users can validate their HED (Hierarchical Event Descriptor) annotations.

6.6.2 Types of errors

HED annotations consist of comma-separated lists of HED tags selected from valid HED vocabularies, referred to as **HED schemas**. HED annotations may include arbitrary levels of parentheses to clarify associations among HED tags.

In some cases, mainly in BIDS sidecars, HED annotations may contain # placeholders, which are replaced by values from the appropriate columns of an associated event file when HED annotations are assembled for analysis.

Two types of errors can occur: **syntactic** and **semantic**.

> **Syntactic** errors refer to format errors that aren't related to any particular HED schema, for example, missing commas or mismatched parentheses.
>
> **Semantic** errors refer to annotations that don't comply with the rules of the particular HED vocabularies used in the annotation, for example, invalid HED tags or values that have the wrong units or type. Semantic errors also include higher-level requirements such as missing definitions or unmatched *Offset* tags when designating the **temporal scope** of events.

Current versions of the validators do not separate these phases and require that the appropriate HED schemas are available at the time of validation.

See **HED validation errors** for a list of the validation errors that are detected by validation tools.

6.6.3 Available validators

HED currently supports native validators for Python and JavaScript. Both validators support **HED-specification v3.0.0**.

6.6.3.1 Python validator

The Python validator included in **HEDTools** on PyPI is used as the basis for most HED analysis tools. Generally, new HED features are first implemented and tested in this validator before propagating to other tools in the HED ecosystem. The source code for HEDTools is available in the **hed-python** GitHub repository. The latest features appear on the `develop` branch before being propagated to `master` and then released.

6.6.3.2 JavaScript validator

The JavaScript **hed-validator** on npm is the package used for validation in **BIDS**. Although the main interface is designed for BIDS integration, the underlying validation functions can be called directly. The source code is available in the **hed-javascript** GitHub repository.

6.6.3.3 MATLAB support

Validation in MATLAB is currently not directly supported, although some discussion about future native support is ongoing. MATLAB users should use the **HED online validation tools** or the ***HED RESTful services interface*** as discussed ***below***.

6.6.4 Validation strategies

In most experiments, the event files and metadata sidecars share a common structure. A practical HED approach is to annotate and validate a single events file and json sidecar using the online tools before trying to validate entire dataset. If most of the annotations are in a BIDS JSON sidecar, this may be all you need to complete annotation.

How to approach HED annotation.

1. Use the online tools to validate a single event file and sidecar if available.
2. Correct errors. (This will get most of the HED errors out.)
3. Use Jupyter notebooks or the remodeling tools to fully validate the HED in the dataset.
4. Use the BIDS validators to validate all aspects of the dataset, if the dataset is in BIDS.

6.6.4.1 Validation in BIDS

BIDS validates many aspects of a dataset beyond HED, including the format and metadata for all the files in the dataset. Thus, a dataset may have valid HED annotations but not be completely valid in BIDS.

Specifying the HED version

BIDS datasets that have HED annotations, should have the `HEDVersion` field specified in `dataset_description.json` as illustrated in the following example:

Sample dataset_description.json for a BIDS dataset.

```
{
    "Name": "Face processing MEEG dataset with HED annotation",
    "BIDSVersion": "1.8.4",
    "HEDVersion": "8.1.0",
    "License": "CC0"
}
```

BIDS online validator

The simplest way to validate a BIDS dataset is to use the BIDS online validator:

_static/images/BIDSOnlineValidator.png

The BIDS online validator is available at **https://bids-standard.github.io/bids-validator/**. The BIDS validators use the **hed-validator** JavaScript package available at **npm** to do the validation.

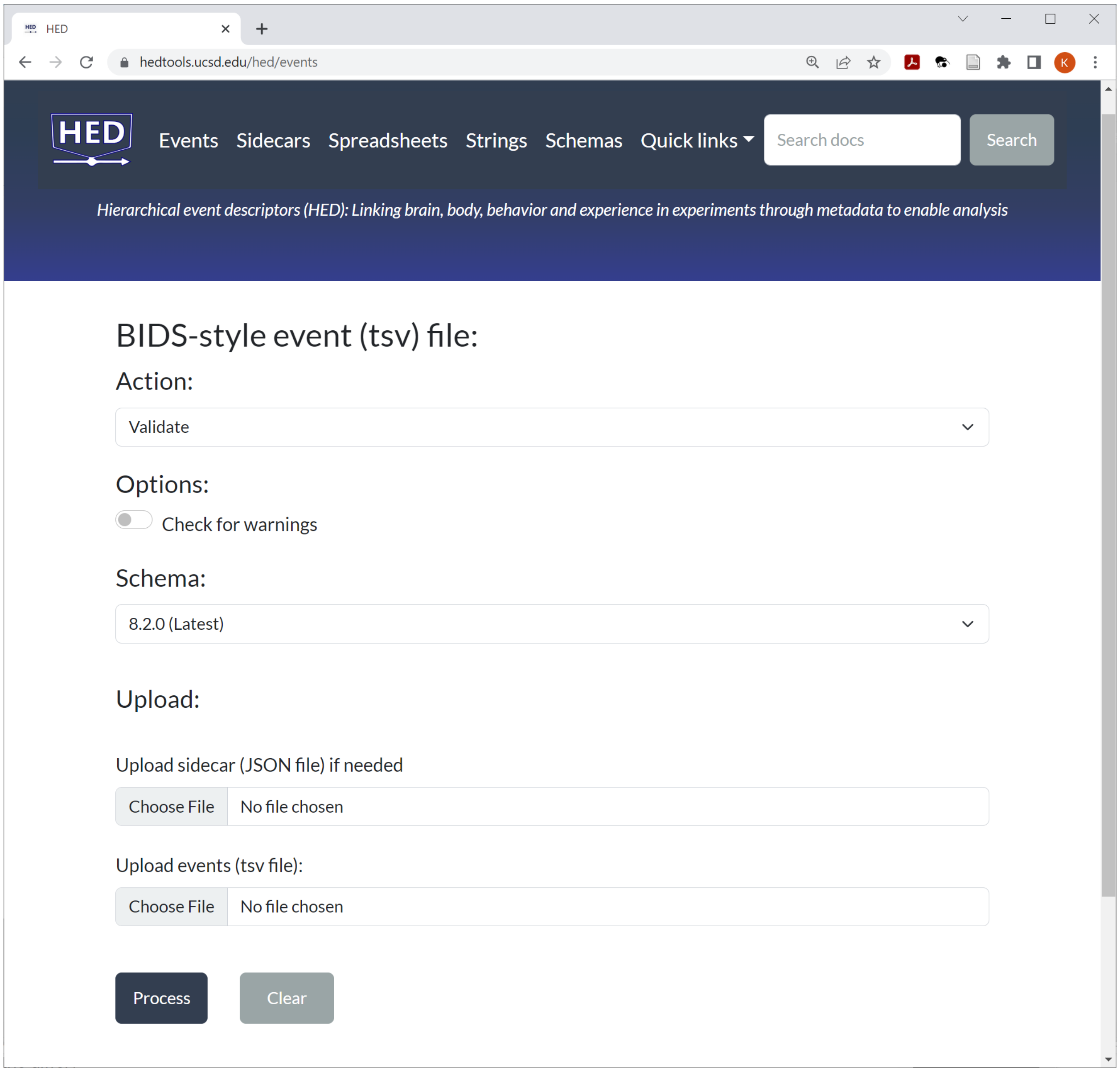
See the **bids-validator** for additional details.

6.6.4.2 HED online validation

The HED online validation tools are available at https://hedtools.ucsd.edu/hed. The HED web-based tools are designed to act on a single file (e.g. events, sidecar, spreadsheet, schema), but may require supporting files.

For example, the following screenshot shows the menu for the online event validation tools. The buttons in the banner allow you to select the type of file to operate on.

Once you have selected the type by pressing the banner button, you will see a menu for the particular type selected, in this case an events file.



Hierarchical event descriptors (HED): Linking brain, body, behavior and experience in experiments through metadata to enable analysis

BIDS-style event (tsv) file:

Action: Validate

Options: Check for warnings

Schema: 8.2.0 (Latest)

Upload:

Upload sidecar (JSON file) if needed: Choose File No file chosen

Upload events (tsv file): Choose File No file chosen

Menu for validating an events file using the HED online tools.

The default action for events is validation, but you can choose other operations by picking another action. The validate operation has one option: whether to check for warnings as well as errors.

Upload the events file and the supporting JSON sidecar using the *Browser* buttons. If you aren't using the latest HED vocabulary, you can use the *HED schema version* pull-down to select the desired schema.

When you press the *Process* button, the files (event file and sidecar) are validated. If the files have errors, a downloadable text file containing the error messages is returned. Otherwise, a message indicating successful validation appears at the bottom of the screen.

The online tools support many other operations and most of them automatically validate the files before applying the requested operation. For example, one of the available actions shown on the menu above is assembling all the HED tags applicable to each line in the events file.

New features of the tools take a while to propagate to the released version of the online tools. Use the **HED online development server** to access the latest versions.

6.6.4.3 Validation for MATLAB users

HED validation in MATLAB is currently done by accessing the HED online tools as web services.

Direct access to services

Users can access these services directly by calling using the HED MATLAB web services functions as explained in ***HED services in MATLAB***. Download the **web_services** directory from GitHub and include this directory in your MATLAB path. The **runAllTests.m** script calls all the services on test data.

Access through EEGLAB

EEGLAB users can access HED validation through the ***EEGLAB HEDTools plugin***.

CTagger is an annotation tool that guides users through the tagging process using a graphical user interface. CTagger is available as a stand-alone program as well from EEGLAB through the HEDtools plugin.

Access through Fieldtrip

An interface for accessing HED in **Fieldtrip** has recently been added, but is not yet fully documented.

6.6.4.4 Validation for Python users

The HEDTools for Python are available on PyPI and can be installed using the usual Python package installation mechanisms with PIP. However, new features are not immediately available in the released version. If you need the latest version you should install the `develop` branch of the GitHub **hed-python** repository directly using PIP.

Installing the Python HedTools from the develop branch on GitHub.

```
pip install git+https://github.com/hed-standard/hed-python/@develop
```

Jupyter notebooks for validation

Several **Jupyter notebooks** are available as wrappers for calling various Python HED tools.

For example, the **bids_validate_datasets.ipynb** notebook shown in the following example validates an entire BIDS dataset just give the path to the root directory of the dataset.

Python code to validate HED in a BIDS dataset.

```
import os
from hed.errors import get_printable_issue_string
from hed.tools import BidsDataset

## Set the dataset location and the check_for_warnings flag
check_for_warnings = False
bids_root_path = 'Q:/PerceptionalON'

## Validate the dataset
bids = BidsDataset(bids_root_path)
issue_list = bids.validate(check_for_warnings=check_for_warnings)
if issue_list:
    issue_str = get_printable_issue_string(issue_list, "HED validation errors: ", skip_
↪filename=False)
else:
    issue_str = "No HED validation errors"
print(issue_str)
```

Errors, if any are printed to the command line.

Remodeling validation summaries

Validation is also available through HED remodeling tool interface. As explained in ***File remodeling quickstart***, the HED remodeling tools allow users to restructure their event files and/or summarize their contents in various ways. Users specify a list of operations in a JSON remodeling file, and the HED remodeler executes these operations in sequence.

Validation is a summary operation, meaning that it does not modify any event files, but rather produces a summary, in this case of HED validation errors for the dataset. An example of a remodeling operation that is not a summary operation is an operation to rename the columns in an event file.

The following example shows a JSON remodel file containing a single operation — validating event files.

Example JSON remodel file for HED validation.

```
[
    {
        "operation": "summarize_hed_validation",
        "description": "Validate event file and list errors in a summary.",
        "parameters": {
            "summary_name": "validate_initial",
            "summary_filename": "validate_initial",
            "check_for_warnings": true
```

(continues on next page)

(continued from previous page)

```
        }
    }
]
```

Since remodeling summaries do not affect the actual contents of the events files, This summary can be created without using the backup infrastructure.

The following example performs HED validation on the BIDS dataset (*-b* option) whose root directory is `/root_path`. The remodeling file path (corresponding to the JSON in the previous example) is also set.

Example JSON remodel file for HED validation.

```python
import hed.tools.remodeling.cli.run_remodel as run_remodel

data_root = "/root_path"
model_path = "/root_path/derivatives/remodel/models/validate_rmdl.json"
args = [data_root, model_path, '-x', 'derivatives', 'stimuli', '-n', '', '-b', '-r', '8.
↪1.0']
run_remodel.main(args)
```

This remodeling action will perform HED validation on all the event files in the specified BIDS dataset, excluding the `derivatives` and `stimuli` directories (*-x* option). The event files and associated sidecars are located using the BIDS naming convention.

The results of the validation are stored in the file name specified in the remodeling file in the `derivatves/remodel/summaries` directory under the data root. A timestamp is appended to the file name each time the operation is executed to distinguish files.

Both a `.json` and a `.txt` file are created. For example, the text file is: `/root_path/derivatives/remodel/summaries/validate_initial_xxx.txt` where xxx is the time of generation.

For more information see ***File remodeling quickstart*** for an overview of the remodeling process and ***File remodeling tools*** for detailed descriptions of the operations that are currently supported.

6.7 HED search guide

Many analysis methods locate event markers with specified properties and extract sections of the data surrounding these markers for analysis. This extraction process is called **epoching** or **trial selection**.

Analysis may also exclude data surrounding particular event markers.

Other approaches find sections of the data with particular signal characteristics and then determine which types of event markers are more likely to be associated with data sections having these characteristics.

At a more global level, analysts may want to locate datasets whose event markers have certain properties in choosing data for initial analysis or for comparisons with their own data.

6.7.1 HED search basics

Datasets whose event markers are annotated with HED (Hierarchical Event Descriptors) can be searched in a dataset independent manner. The HED search facility has been implemented in the Python **HEDTools** library, an open source Python library. The latest versions are available in the **hed-python** GitHub repository.

To perform a query using HEDTools, users create a query object containing the parsed query. Once created, this query object can then be applied to any number of HED annotations – say to the annotations for each event-marker associated with a data recording.

The query object returns a list of matches within the annotation. Usually, users just test whether this list is empty to determine if the query was satisfied.

6.7.1.1 Calling syntax

To perform a search, create a `TagExpressionParser` object, which parses the query. Once created, this query object can be applied to search multiple HED annotations. The syntax is demonstrated in the following example:

Example calling syntax for HED search.

```
schema = load_schema_version("8.1.0")
hed_string = HedString("Sensory-event, Sensory-presentation", schema=schema)
query_string = "Sensory-event"
query = QueryParser(query_string)
result = query.search(hed_string)
if result:
    print(f"{query_string} found in {str(hed_string)})
```

In the example the strings containing HED annotations are converted to a `HedString` object, which is a parsed representation of the HED annotation. The query facility assumes that the annotations have been validated. A `HedSchema` is required. In the example standard schema version 8.1.0 is loaded. The schemas are available on GitHub.

The query is represented by a `QueryParser` object. The `search` method returns a list of groups in the HED string that match the query. This return list can be quite complex and usually must be filtered before being used directly. In many applications, we are not interested in the exact groups, but just whether the query was satisfied. In the above example, the `result` is treated as a boolean value.

Warning:

- If you are searching many strings for the same expression, be sure to create the `QueryParser` only once.
- The current search facility is case-insensitive.

6.7.1.2 Single tag queries

The simplest type of query is to search for the presence of absence of a single tag. HED offers four variations on the single tag query as summarized in the following table.

Query type	Example query	Matches	Does not match
Single-term Match the term or any child.Don't consider values orextensions when matching.	*Agent-trait*	*Agent-traitAgeAge/35Right-handedAgent-trait/GlassesAgent-property/Agent-trait(Age, Blue)*	*Agent-property*
Quoted-tag Match the exact tag withextension or value	"*Age*"	*AgeAgent-trait/Age*	*Age/35*
	"*Age/34*"	*Age/34Agent-trait/Age/34*	*Age/35*
Tag-path with slash Match the exact tag withextension or value	*Age/34*	*Age/34*	*AgeAge/35Agent-trait/Age/34*
Tag-prefix with wildcard Match the starting portionof a tag and possibly itsvalue or extension.	Age/3*	*Age/34Age/3Agent-trait/Age/34*	*AgeAge/40*

The meanings of the different queries are explained in the following subsections.

Single-term search

In a single-term search, the query is a single term or node in the **HED schema**. The query may not contain any slashes or wildcards.

Single-term queries leverage the HED hierarchical structure, recognizing that schema children of the query term should also satisfy the query. This is HED's **is-a** principle.

The example query in the above table is *Agent-trait*. The full path of *Agent-trait* in the HED schema is *Property/Agent-property/Agent-trait*. Further, the *Agent-trait* has several child nodes including: *Age*, *Agent-experience-level*, *Gender*, *Sex*, and *Handedness*.

The single-term query matches child tags without regard to tag extension or value. Hence, *Agent-trait* matches *Age* which is a child and *Age/35* which is child with a value. *Agent-trait*, itself, may be extended, so *Agent-trait* also matches *Agent-trait/Glasses*. Here *Glasses* is a user-extension.

Quoted-tag search

If the tag-term is enclosed in quotes, the search matches that tag exactly. If you want to match a value as well, you must include that specific value in the quoted tag-term. This is exactly the same as Tag-path with slash, except you can search a single term without a slash.

Tag-path with slash

If the query includes a slash in the tag path, then the query must match the exact value with the slash. Thus, *Age/34* does not match *Age* or *Age/35*. The query matches *Agent-trait/Age/34* because the short-form of this tag-path is *Age/34*. The tag short forms are used for the matching to assure consistency.

Tag-prefix with wildcard

Matching using a tag prefix with the * wildcard, matches the starting portion of the tag. Thus, Age/3* matches *Age/3* as well as *Age/34*.

Notice that the query Age* matches a myraid of tags including *Agent*, *Agent-state*, and *Agent-property*.

6.7.1.3 Logical queries

In the following `A` and `B` represent HED expressions that may contain multiple comma-separated tags and parenthesized groups. `A` and `B` may also contain group queries as described in the next section. The expressions for `A` and `B` are each evaluated and then combined using standard logic.

Query form	Example query	Matches	Does not match
A, B Match if both `A` and `B`are matched.	*Event*, *Sensory-event*	*Event*, *Sensory-eventSensory-event*, *Event*(*Event*, *Sensory-event*)	*Event*
A and B Match if both `A` and `B` are matched. Same as the comma notation.	*Event* and *Sensory-event*	*Event*, *Sensory-eventSensory-event*, *Event*(*Event*, *Sensory-event*)	
A or B Match if either `A` or `B`.	*Event* or *Sensory-event*	*Event*, *Sensory-eventSensory-event*, *Event*(*Event*, *Sensory-event*)*EventSensory-event*	*Agent-trait*
~AMatch groups that donot contain `A` `A` can be an arbitrary expression.	[[*Event*, *~Action*]]	(*Event*)(*Event*, *Animal-agent*)(*Sensory-event*, (*Action*))	*EventEvent*, *Action*(*Event*, *Action*)
@AMatch a line that does not contain `A`.	*@Event*	*ActionAgent-traitAction*, *Agent-Trait*(*Action*, *Agent*)	*Event*(*Action*, *Event*)(*Action*, *Sensory-event*)(*Agent*, (*Sensory-event*, *Blue*))

6.7.1.4 Group queries

Tag grouping with parentheses is an essential part of HED annotation, since HED strings are independent of ordering of tags or tag groups at the same level.

Consider the annotation:

> *Red*, *Square*, *Blue*, *Triangle*

In this form, tools cannot distinguish which color goes with which shape. Annotators must group tags using parentheses to make the meaning clear:

> (*Red*, *Square*), (*Blue*, *Triangle*)

Indicates a red square and a blue triangle. Group queries allow analysts to detect these groupings.

As with logical queries, A and B represent HED expressions that may contain multiple comma-separated tags and parenthesized groups.

Query form	Example query	Matches	Does not match
[[A, B]]Match a group thatcontains both A and Bat the same levelin the same group.	*[[Red, Blue]]*	*(Red, Blue)(Red, Blue, Green)*	*(Red, (Blue, Green))*
[A, B] Match a group thatcontains A and B. Both A and B couldbe any subgroup level.	*[Red, Blue]*	*(Red, (Blue, Green))((Red, Yellow), (Blue, Green))*	*Red, (Blue, Green)*

These operations can be arbitrarily nested and combined, as for example in the query:

[A or [[B and C]]]

In this query Ordering on either the search terms or strings to be searched doesn't matter unless it will impact precedence on the expression. Use logical grouping with parentheses to assure the expected order.

Precedence is purely left to right outside of grouping operations. Thus, unlike many traditional programming languages, **and** does not take precedence over **or**. This may change in the future.

6.7.2 Where can HED search be used?

The HED search facility allows users to form sophisticated queries based on HED annotations in a dataset-independent manner. These queries can be used to locate data sets satisfying the specified criteria and to find the relevant event markers in that data.

For example, the **factor_hed_tags** operation of the HED **file remodeling tools** creates factor vectors for selecting events satisfying general HED queries.

The **HED-based epoching** tools in **EEGLAB** can use HED-based search to epoch data based on HED tags.

Work is underway to integrate HED-based search into other tools including **Fieldtrip** and **MNE-python** as well into the analysis platforms **NEMAR** and **EEGNET**

6.8 HED summary guide

The HED **File remodeling tools** provide a number of event summaries and event file transformations that are very useful during curation and analysis.

The summaries described in this guide are:

- ***Column value summary***
- ***HED tag summary***
- ***Experimental design summary***

As described in more detail in the **File remodeling quickstart** tutorial and the **File remodeling tools** user manual, these tools have as input, a JSON file with a list of remodeling commands and an event file. Summaries involving HED also require a HED schema version and possibly a JSON sidecar containing HED annotations.

The summary tools produce text and/or JSON summaries of the tabular files (usually event files). Summaries accumulate the results for each tabular file that is input. When the results are output, the summary tools produce an overall summary of all input files that have been processed and, if requested, also include an individual summary for each input file.

The examples in this tutorial use the Wakeman-Hanson Face Processing dataset as an example. A **reduced version** containing 2 subjects and no imaging data is used to produce the summaries in the examples. The reduced dataset has 6 event files each containing 200 events. The full dataset is available on OpenNeuro as **ds003645**.

Each example only shows the overall summary with links to the full summaries that include individual summaries. The summaries use a **[number events, number files]** display of the counts of how many events and files an item appears in.

6.8.1 Column value summary

The `summarize_column_value` operation produces a summary of three types of columns:

- **categorical column**: the summary counts the number of events (rows) and files for each unique column value.
- **value column**: the summary counts the number of files containing the column and total number of rows in the column.
- **skip columns**: are ignored.

The categorical column information is useful for spotting inconsistencies and unexpected values. For example, if a trial consists of **stimulus–>key-press–>feedback** and there are fewer key-press events in a file than stimulus or feedback events, you can conclude that either the participant failed to respond in some trials or the responses were not properly recorded.

The value column information in the current release of the remodeling tools is limited. However, if a value column file count is different from the number of event files in the dataset, you can conclude that some event files are missing that column or have that column multiple times. More extensive information reporting for value columns is planned for future releases.

A sample JSON remodeling file with the command for creating a column value summary is shown in the following example. The remodeling file specifies how columns are treated. Columns that are not listed as *skip_columns* or *value_columns* are assumed to be categorical columns.

Example JSON remodeling file for a column value summary.

```
[{
   "operation": "summarize_column_values",
   "description": "Summarize the column values in an excerpt.",
   "parameters": {
       "summary_name": "column_values_summary",
       "summary_filename": "column_values_summary",
       "skip_columns": ["onset", "duration"],
       "value_columns": ["stim_file", "trial"]
   }
}]
```

The following excerpt shows the dataset portion of the resulting summary in text format:

Text format excerpt with dataset-level summary of column values.

```
Context name: column values summary
Context type: column_values
Context filename: column_values_summary
```

(continues on next page)

(continued from previous page)

```
Overall summary:
Dataset: Total events=1200 Total files=6
   Categorical column values[Events, Files]:
      event_type:
         double_press[1, 1] left_press[83, 4] right_press[168, 6] setup_right_sym[6, 6]␣
↪show_circle[316, 6] show_cross[310, 6] show_face[310, 6] show_face_initial[6, 6]
      face_type:
         famous_face[108, 6] n/a[884, 6] scrambled_face[103, 6] unfamiliar_face[105, 6]
      rep_lag:
         1[77, 6] 10[15, 6] 11[13, 5] 12[9, 5] 13[7, 6] 14[6, 4] 15[2, 2] 6[1, 1] 7[2,␣
↪2] 8[6, 4] 9[10, 6] n/a[1052, 6]
      rep_status:
         delayed_repeat[71, 6] first_show[168, 6] immediate_repeat[77, 6] n/a[884, 6]
   Value columns[Events, Files]:
      stim_file[1200, 6]
      trial[1200, 6]
```

Notice that there is one *double_press* event in the *event_type* column of one of the six event files analyzed in this summary. To narrow down which file this *double_press* event occurred in, we could look at the **full text summary**, which includes individual summaries for each event file.

We also observe that three values *famous_face*, *unfamiliar_face* and *scrambled_face* appear roughly the same number of times in the *face_type* across the six dataset. The large number of *n/a* values in *face_type* is because the type of face is only specified for the stimulus events:

108 (*famous_face*) + 103 (*scrambled_face*) + 105 (*unfamiliar_face*) =
310 (*show_face*) + 6 (*show_face_initial*)

As expected, the *show_face_initial* appears exactly once in each file (e.g., [6 events, 6 files]) since it is a setup-event.

6.8.2 HED tag summary

The HED tag summary gives an overall picture of the types of HED tags in the dataset along with counts and the number of files that these tags appear in. An advantage that HED tag summaries have over straight column value summaries is that the tags are comparable across experiments, while column values are experiment-specific.

The *tags* dictionary specifies how the results should be reported. In the following remodeling file for generating a HED tag summary, the tag counts will be grouped under the titles: “Sensory events”, “Agent actions” and “Items”. Tags that don’t fit in these three categories will be grouped under “Other tags”.

Example JSON remodeling file for a HED tag summary.

```
[{
   "operation": "summarize_hed_tags",
   "description": "Summarize the HED tags in the dataset.",
   "parameters": {
       "summary_name": "hed_tag_summary",
       "summary_filename": "hed_tag_summary",
       "tags": {
           "Sensory events": ["Sensory-event", "Sensory-presentation",
                              "Task-stimulus-role", "Experimental-stimulus"],
           "Agent actions": ["Agent-action", "Agent", "Action", "Agent-task-role",
```

(continues on next page)

(continued from previous page)

```
                              "Task-action-type", "Participant-response"],
            "Objects": ["Item"]
          },
        "expand_context": false
    }
}]
```

The following excerpt shows the dataset portion of the resulting summary in text format when running on the **reduced version** face processing dataset, which has 6 event files containing a total of 1200 events.

Text format excerpt with dataset-level summary of hed tag counts

```
Context name: summarize_hed_tags
Context type: hed_tag_summary
Context filename: hed_tag_summary

Dataset
    Main tags[events,files]:
        Sensory events:
            Sensory-event[942,6] Cue[626,6] Experimental-stimulus[316,6]
        Agent actions:
            Agent-action[252,6] Press[1,1] Indeterminate-action[1,1] Participant-
↪response[251,6]
        Objects:
            Image[942,6] Face[148,6] Keyboard-key[1,1]
    Other tags[events,files]:
        Experiment-structure[6,6] Def[1199,6] Onset[948,6] Experimental-trial[1194,6]
        Pathname[942,6] Intended-effect[626,6] Offset[936,6] Item-interval[148,6]
```

The summary indicates that the event type breakdown:

942 sensory events + 252 agent actions + 6 experiment structure events = 1200 events

Further, there were 626 cues and 316 experimental stimuli among the sensory events.

6.8.3 Experimental design summary

The HED type summary allows users to obtain a detailed summary of a particular tag. Usually type summaries are used for *Condition-variable* tag, which encodes experimental conditions and design. The ***HED conditions and design matrices*** tutorial explains how this information is encoded and can be used.

Type summaries based on the *Task* tag and *Time-block* tag are also informative.

Example JSON remodeling file for a HED type summary based on *Condition-variable*.

```
[{
   "operation": "summarize_hed_type",
   "description": "Summarize the condition variable tags in the dataset.",
   "parameters": {
       "summary_name": "wh_condition_variables",
```

(continues on next page)

(continued from previous page)

```
        "summary_filename": "wh_condition_variables",
        "type_tag": "condition-variable"
     }
}]
```

The HED type summaries automatically expand the event-context, so that an event that has an *Onset* tag will affect all intermediate events until its *Offset*.

The result of applying the above operation to the sample data is:

Text format excerpt with dataset-level summary of hed type (condition-variable) counts.

```
Dataset: 3 condition-variable types in 6 files with a total of 1200
   key-assignment: 1 levels in 1200 events out of 1200 total events in 6 files
      right-sym-cond [1200 events, 6 files]:
         Tags: ['Index-finger', 'Right-side-of', 'Experiment-participant', 'Behavioral-
↪evidence',
                'Symmetrical', 'Index-finger', 'Left-side-of', 'Experiment-participant',
                'Behavioral-evidence', 'Asymmetrical']
         Description: Right index finger key press indicates a face with above average␣
↪symmetry.
   face-type: 3 levels in 316 events out of 1200 total events in 6 files
      unfamiliar-face-cond [105 events, 6 files]:
         Tags: ['Image', 'Face', 'Unfamiliar']
         Description: A face that should not be recognized by the participants.
      famous-face-cond [108 events, 6 files]:
         Tags: ['Image', 'Face', 'Famous']
         Description: A face that should be recognized by the participants
      scrambled-face-cond [103 events, 6 files]:
         Tags: ['Image', 'Face', 'Disordered']
         Description: A scrambled face image generated by taking face 2D FFT.
   repetition-type: 3 levels in 316 events out of 1200 total events in 6 files
      first-show-cond [168 events, 6 files]:
         Tags: ['Item-count', 'Face', 'Item-interval']
         Description: Factor level indicating the first display of this face.
      immediate-repeat-cond [77 events, 6 files]:
         Tags: ['Item-count', 'Face', 'Item-interval']
         Description: Factor level indicating this face was the same as previous one.
      delayed-repeat-cond [71 events, 6 files]:
         Tags: ['Item-count', 'Face', 'Item-interval', 'Greater-than-or-equal-to', 'Item-
↪interval']
         Description: Factor level indicating face was seen 5 to 15 trials ago.
```

This summary has three condition variables: *key-assignment*, *face-type* and *repetition-type*. The *face-type* and *repetition-type* each have three levels encoding a 3 x 3 experimental design. The *face-type* condition variable has three levels with roughly equal numbers of occurrences (*famous-face-cond* with 108 events, *scrambled-face-cond* with 103 events, and *unfamiliar-face-cond* with 105 events).

This information is similar to that obtained in the ***column value summary***, but only because these condition variables were directly encoded by columns `face_type` and `repetition_type` in the events files. The HED approach allows a more general, dataset-independent extraction of design matrices and experimental conditions.

The final condition variable *key-assignment* only has one level and appears in all events in all the files. In reality the key assignment is designated in a single event in each file, but it appears with an *Onset* and no *Offset*, indicating that it runs until the end of the file. The *key-assignment* condition actually has two levels: *right-sym-cond* and *left-sym-cond*, but this condition is counter-balanced across subjects rather than trials. The two subjects in the sample data both were assigned the *right-sym-cond*.

6.9 HED conditions and design matrices

This tutorial discusses how information from neuroimaging experiments should be stored and annotated so that the underlying experimental design and experimental conditions for a dataset can be automatically extracted, summarized, and used in analysis. The mechanisms for doing this use HED (Hierarchical Event Descriptors) in conjunction with a BIDS (Brain Imaging Data Structure) representation of the dataset.

The tutorial assumes that you have a basic understanding of HED and how HED annotations are used in BIDS. Please review **Annotating a BIDS dataset**, the **BIDS annotation quickstart**, and the **HED annotation quickstart** tutorials as needed.

The ***Experimental design concepts*** section at the end of this tutorial provides a basic introduction to the ideas of factor vectors and experimental design if you are unfamiliar with these topics.

- ***HED annotations for conditions***
  - ***Direct condition variables***
  - ***Defined condition variables***
  - ***Direct vs defined approaches***)
  - ***Column vs row annotations***
- ***Experimental design concepts***
  - ***Design matrices and factor variables***
  - ***Types of condition encoding***

This tutorial introduces tools and strategies for encoding information about the experimental design as part of a dataset metadata without excessive effort on the part of the researcher. The discussion mainly focuses on categorical variables.

6.9.1 HED annotations for conditions

As mentioned above, HED (Hierarchical Event Descriptors) provide several mechanisms for easily annotating the experimental conditions represented by a BIDS dataset so that the information can be automatically extracted, summarized, and used by tools.

HED has three ways of annotating experimental conditions: condition variables without definitions, condition variables with definitions but no levels, and condition variables with levels. All three mechanisms use the *Condition-variable* tag as part of the annotation. The three mechanisms can be used in any combination to document the experimental design for a dataset.

6.9.1.1 Direct condition variables

The simplest way to encode experimental conditions is to use named *Condition-variable* tags for each condition value. The following is a sample excerpt from a simplified event file for an experiment to distinguish brain responses for houses and faces.

Example 1. Excerpt from a sample event file from a simplified house-face experiment.

onset	duration	event_type	stim_file
2.010	0.1	show_house	ranch1.png
3.210	0.1	show_house	colonial68.png
4.630	0.1	show_face	female43.png
6.012	0.1	show_house	castle2.png
7.440	0.1	show_face	male81.png

As explained in **BIDS annotation quickstart**, the most commonly used strategy for annotating events in a BIDS dataset is to create a single JSON file located in the dataset root containing the annotations for the columns. The following shows a minimal example:

Example 2: Minimal JSON sidecar with HED annotations for Example 1.

```
{
   "event_type": {
      "HED": {
         "show_house": "Sensory-presentation, Visual-presentation, Experimental-stimulus,
↪ (Image, Building/House), Condition-variable/House-cond",
         "show_face": "Sensory-presentation, Visual-presentation, Experimental-stimulus,␣
↪(Image, Face), Condition-variable/Face-cond"
      }
   },
   "stim_file": {
      "HED": "(Image, Pathname/#)"
   }
}
```

Each row in an `events.tsv` file represents a time marker in the corresponding data recording. At analysis time, HED tools look up each `events.tsv` column value in the JSON file and concatenate the corresponding HED annotation into a single string representing the annotation for that row. Annotations without #'s are used directly, while annotations with # have the corresponding column values substituted when the annotation is assembled.

Example 3 shows the Hed annotation for the first row in the `events.tsv` file of Example 1.

Example 3: HED annotation for first event in Example 1 using JSON sidecar of Example 2.

> "*Sensory-presentation*, *Visual-presentation*, *Experimental-stimulus*,
> (*Image*, *Building/House*), *Condition-variable/House-cond*,
> (*Image*, *Pathname/ranch1.png*)"

Notice that *Building/House* is a partial path rather than a single tag. This is because *House* is currently not part of the base HED vocabulary. However, users are allowed to extend tags at most nodes in the HED schema, but they must use a path that includes a least one ancestor in the HED schema.

HED tools have the capability of automatically detecting *Condition-variable* tags in annotated HED datasets to create factor vectors and summaries automatically. Example 4 shows the event file after HED tools have appended one-hot factor vectors for the two condition variables *Condition-variable/House-cond* and *Condition-variable/Face-cond*. The 1's and 0's *house_cond* and *face-cond* columns indicate presence or absence of the corresponding condition variables.

Example 4. Event file from Example 2 after one-hot factor vector extraction.

onset	duration	event_type	stim_file	house-cond	face-cond
2.010	0.1	show_house	ranch1.png	1	0
3.210	0.1	show_house	colonial68.png	1	0
4.630	0.1	show_face	female43.png	0	1
6.012	0.1	show_house	castle2.png	1	0
7.440	0.1	show_face	male81.png	0	1

Example 5 shows a JSON summary that HED tools can extract from a single events file once a dataset has been annotated using HED. This very simple example only had two condition variables and only used direct references to these condition variables. Dataset-wide summaries can also be extracted.

Example 5: The HED tools summary of condition variables for Example 4.

```
{
   "house-cond": {
      "name": "house-cond",
      "variable_type": "condition-variable",
      "levels": 0,
      "direct_references": 3,
      "total_events": 5,
      "number_type_events": 3,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {}
   },
   "face-cond": {
      "name": "face-cond",
      "variable_type": "condition-variable",
      "levels": 0,
      "direct_references": 2,
      "total_events": 5,
      "number_type_events": 2,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {}
   }
}
```

The summary shows that of the 5 events in the file: 3 events were under the house condition and 2 events were under the face condition. There were no events in multiple categories of the same condition variables (which would not be possible since these condition variables were referenced directly rather than using assigned levels). All names are translated to lower case as HED is case-insensitive with respect to analysis, and the summary and factorization tools convert to lower case before processing.

These HED summaries can be created for other tags besides *Condition-variable*, hence the *variable_type* is given in the summary of Example 5. Other commonly created summaries are for *Task* and *Control-variable*.

In this example, the two conditions: *house-cond* and *face-cond* are treated as though they were unrelated. These direct condition variables are very easy to annotate— just make up a name and stick the tags anywhere you want to create factor variables or summaries. However, a more common situation is for a condition variable to have multiple levels, which direct use condition variables does not support.

Another disadvantage of direct condition variables is that there is no information about what the conditions represent beyond the arbitrarily chosen condition names.

A third disadvantage is that direct condition variables can not be used to anchor events with temporal extent.

The next section introduces defined condition variables, which address both of these disadvantages.

6.9.1.2 Defined condition variables

Example 6: A revised JSON sidecar using defined conditions for Example 1.

```
{
   "event_type": {
      "HED": {
         "show_house": "Sensory-presentation, Visual-presentation, Experimental-stimulus,
↪ (Image, Building/House), Def/House-cond",
         "show_face": "Sensory-presentation, Visual-presentation, Experimental-stimulus,␣
↪(Image, Face), Def/Face-cond"
      }
   },
   "stim_file": {
      "HED": "(Image, Pathname/#)"
   },
   "my_definitions": {
      "HED": {
          "house_cond_def": "(Definition/House-cond, (Condition-variable/Presentation-
↪type, (Image, Building/House)))",
          "face_cond_def": "(Definition/Face-cond, (Condition-variable/Presentation-type,
↪ (Image, Face)))"
}
```

Example 6 defines a condition variable called *Presentation-type* with two levels: *House-cond* and *Face-cond*. The definitions of *House-cond* and *Face-cond* both include the same *Presentation-type Condition-variable* so tools recognize these as levels of the same variable and automatically extract the 2-factor experimental design.

Notice that the (*Image*, *Building/House*) tags are included both in the definition of the *House-cond* level of the *Presentation-type* condition variable and in the tags for the *event_type* column value *show_house*. Similarly, the (*Image*, *Face*) tags appear in both the definition of the *Face-cond* level of the *Presentation-type* condition variable and in the tags for the *event_type* column value *show_face*. We have included these tags in both places because generally the condition variable definitions are removed prior to searching for HED tags. The tags in the definitions define the meaning of the conditions.

Example 7: The summary extracted when the JSON sidecar of Example 6 is used.

```
{
   "presentation-type": {
      "name": "presentation-type",
      "variable_type": "condition-variable",
      "levels": 2,
      "direct_references": 0,
      "total_events": 5,
      "number_type_events": 5,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {
         "house-cond": 3,
         "face-cond": 2
      }
   }
}
```

6.9.1.3 Direct vs defined approaches

Table 1 compares the two approaches for encoding experimental conditions and design in HED. Both approaches use the *Condition-variable* tag. While direct condition variables (just using a *Condition-variable* tag without defining it) is very easy, it provides limited information about meaning in downstream summaries. In general defined condition variables, while more work, provide a more complete picture.

Table 1: **Table 1:** Comparison of direct versus definition conditions.

Approach	Advantages	Disadvantages
Direct	Easy to use–just a label.Can appear in summaries.Can generate factor vectors.	Give no information about meaning.No levels for condition variables.Limited information about experimental design.Do not support event temporal extent.
Defined	Better information in summaries.Encode condition variables with levels.Can give factor vectors for levels.Better experimental design information.Can anchor events with temporal extent.	Must give definitions.

It should be noted that other tags, particularly those in the HED `Structural-property` subtree such as `Task` can be summarized and used as factor vectors in a way similar to *Condition-variable*.

6.9.1.4 Column vs row annotations

In this section, we look at a more complicated example based on the Wakeman-Henson face-processing dataset. This dataset, which is available on OpenNeuro under accession number ds003645, was used in as a case study on HED annotation described in the Capturing the nature of events paper. The experiment is based on a 3 x 3 x 2 experimental design: face type x repetition status x key choice.

The experimental stimulus in each trial was the visual presentation of one of 3 possible types of images: a well-known face, an unfamiliar face, and a scrambled face image. The type of face was randomized across trials.

The repetition status condition variable also had one of three possible values and indicated whether the stimulus image had not been seen before (first show), was just seen in the previous trial (immediate repeat), or had been last seen several trials ago (delayed repeat). The repetition status was randomized across trials.

The final condition variable in the experimental design was the key assignment. In the right symmetry condition, participants pressed the right mouse button to indicate that the presented face had above average symmetry, while in the left symmetry condition, participants pressed the left mouse button to indicate that the presented face had above average symmetry. The key assignment was held constant for each recording, but the key choice was counter-balanced across participants.

Example 8 shows an excerpt from the event file of sub-002 run-1. (You may find it useful to look at the full event file sub-002_task-FacePerception_run-1_events.tsv and the dataset's JSON sidecar with full HED annotations: task-facePerception_events.json

Example 8: An excerpt from the Wakeman-Henson face-processing dataset.

onset	duration	event_type	face_type	rep_status	trial	rep_lag	value	stim_file
0.004	n/a	setup_right_sym	n/a	n/a	n/a	n/a	3	n/a
24.2098	n/a	show_face_initial	unfamiliar_face	first_show	1	n/a	13	u032.bmp
25.0353	n/a	show_circle	n/a	n/a	1	n/a	0	circle.bmp
25.158	n/a	left_press	n/a	n/a	1	n/a	256	n/a
26.7353	n/a	show_cross	n/a	n/a	2	n/a	1	cross.bmp
27.2498	n/a	show_face	unfamiliar_face	immediate_repeat	2	1	14	u032.bmp
27.8971	n/a	left_press	n/a	n/a	2	n/a	256	n/a
28.0998	n/a	show_circle	n/a	n/a	2	n/a	0	circle.bmp
29.7998	n/a	show_cross	n/a	n/a	3	n/a	1	cross.bmp
30.3571	n/a	show_face	unfamiliar_face	first_show	3	n/a	13	u088.bmp

Example 8 illustrates two different ways of using defined conditions for encoding: **inserting an event with temporal extent** or using **column encoding**.

The key assignment condition is marked by inserting an event with *event_type* equal to *setup_right_sym* at the beginning of the file. As we shall see below, this event is annotated with having temporal extent, as defined by HED *Onset* and *Offset* tags, so the condition is in effect until the event's extent ends.

In the column strategy, an event file column represents the condition variable, and the values in that column represent the levels. With this encoding, the condition variable is only applicable at a particular level when that level name appears in the column. An n/a value in that column indicates the condition does not apply to that event.

Example 9 shows the portion of the **task-facePerception_events.json** that encodes information about the *setup_right_sym* event found as the first event in the event file excerpt of Example 8. This excerpt only contains the relevent definition and the relevant annotation.

Example 9: Excerpt of the JSON sidecar relevant to the *setup_right_sym* event.

```
{
   "event_type": {
      "HED": {
         "setup_right_sym": "Experiment-structure, (Def/Right-sym-cond, Onset), (Def/
```

(continues on next page)

(continued from previous page)

```
↪Initialize-recording, Onset)"
      }
   },
   "hed_def_conds": {
      "HED": {
        "right_sym_cond_def": "(Definition/Right-sym-cond, (Condition-variable/Key-
↪assignment, ((Index-finger, (Right-side-of, Experiment-participant)), (Behavioral-
↪evidence, Symmetrical)), ((Index-finger, (Left-side-of, Experiment-participant)),␣
↪(Behavioral-evidence, Asymmetrical)), Description/Right index finger key press␣
↪indicates a face with above average symmetry.))"
      }
   }
}
```

Only the *event_type* column is relevant for assembling the annotations for the first row of Example 8, since the other annotated columns have n/a values. The assembled HED annotation for the first row of Example 8 is shown in Example 10.

Example 10: The HED annotation of the first row of Example 8.

"*Experiment-structure*, (*Def/Right-sym-cond*, *Onset*), (*Def/Initialize-recording*, *Onset*)"

HED represents events of temporal extent using HED definitions with the *Onset* and *Offset* tags grouped with a user-defined term. The (*Def/Right-sym-cond*, *Onset*) specifies that an event defined by *Right-sym-cond* begins at the time-marker represented by this row in the event file. This event continues until the end of the file or until an event marker with (*Def/Right-sym-cond*, *Offset*) occurs. In this case, no *Offset* marker for *Right-sym-cond* appears in the file, so the event represented by *Right-sym-cond* occurs over the entire recording.

The user-defined term is prefixed with *Def/* and indicates what the event of temporal extent represents. If the definition includes a *Condition-variable*, then the event represents the occurrence of that experimental condition. The user-defined terms are usually defined in the `events.json` file located at the top-level of the BIDS dataset.

Example 11 shows a more readable form for the definition of *Right-sym-cond*.

Example 11: The contents of the definition for *Right-sym-cond*.

```
(
   Definition/Right-sym-cond, (
      Condition-variable/Key-assignment,
      ((Index-finger, (Right-side-of, Experiment-participant)), (Behavioral-evidence,␣
↪Symmetrical)),
      ((Index-finger, (Left-side-of, Experiment-participant)), (Behavioral-evidence,␣
↪Asymmetrical)),
      Description/Right index finger key press indicates a face with above average␣
↪symmetry.
   )
)
```

The primary use of the definitions for condition variables is to encode the experimental design in an actionable format. Thus, as a general practice, *Defs* representing condition variables are removed prior to searching for other tags to avoid repeats. Notice that both *Right-side-of* and *Left-side-of* appear in the definition. Thus, if these *Defs* were included, every event would have both left and right tags.

Once a dataset includes the appropriate annotations, HED tools can automatically extract the experimental design. Example 12 shows the result of extraction of categorical factor vectors for the event file of Example 8.

Example 12: HED tools categorical form extraction of the design matrix for Example 8.

onset	key-assignment	face-type	repetition-type
0.004	right-sym-cond	n/a	n/a
24.2098	right-sym-cond	unfamiliar-face-cond	first-show-cond
25.0353	right-sym-cond	n/a	n/a
25.158	right-sym-cond	n/a	n/a
26.7353	right-sym-cond	n/a	n/a
27.2498	right-sym-cond	unfamiliar-face-cond	immediate-repeat-cond
27.8971	right-sym-cond	n/a	n/a
28.0998	right-sym-cond	n/a	n/a
29.7998	right-sym-cond	n/a	n/a
30.3571	right-sym-cond	unfamiliar-face-cond	first-show-cond

In the categorical representation, HED uses the condition variable name as the column name. The level values appear in the columns for event markers at which the condition variable at that level applies. Notice that *right-sym-cond* appears in every row because it was used in an event that extended to the end of the file. On the other hand, the other condition variables were encoded using columns and only appear when present in the column.

Note that if an event has multiple levels of the same condition, categorical and ordinal encoding cannot be used. Only one-hot encoding supports multiple levels in the same event.

Example 13 below shows the condition variable summary that HED produces for the full sub-002_task-FacePerception_run-1_events.tsv and JSON sidecar task-facePerception_events.json.

Example 13: The condition variable summary extracted from the full event file.

```
{
   "key-assignment": {
      "name": "key-assignment",
      "variable_type": "condition-variable",
      "levels": 1,
      "direct_references": 0,
      "total_events": 552,
      "number_type_events": 552,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {
         "right-sym-cond": 552
      }
   },
   "face-type": {
      "name": "face-type",
      "variable_type": "condition-variable",
      "levels": 3,
      "direct_references": 0,
```

(continues on next page)

(continued from previous page)

```
      "total_events": 552,
      "number_type_events": 146,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {
         "unfamiliar-face-cond": 47,
         "famous-face-cond": 49,
         "scrambled-face-cond": 50
      }
   },
   "repetition-type": {
      "name": "repetition-type",
      "variable_type": "condition-variable",
      "levels": 3,
      "direct_references": 0,
      "total_events": 552,
      "number_type_events": 146,
      "number_multiple_events": 0,
      "multiple_event_maximum": 1,
      "level_counts": {
         "first-show-cond": 75,
         "immediate-repeat-cond": 36,
         "delayed-repeat-cond": 35
      }
    }
}
```

The file has 552 events. Since the *key-assignment* condition variable with level *right-sym-cond* applies to every event in this file, the *number_type_events* is also 552. On the other hand, the *face-type* condition variable is only applicable in 146 events.

All the condition variables have *number_multiple_events* equal to 0, so any of the three possible encodings: categorical, ordinal, or one-hot can be used.

6.9.2 Experimental design concepts

Traditional neuroimaging experiments are carefully designed to control and document the external conditions under which the experiment is conducted. Often a few items such as the task or the properties of a stimulus are systematically varied as the stimulus is presented and participant responses are recorded.

For example, in an experiment to test for differences in brain responses to pictures of houses versus pictures of faces, the researcher would label time points in the recording corresponding to presentations of the respective pictures so that differences in brain responses between the two types of pictures could be observed. An fMRI analysis might determine which brain regions showed a significant response differential between the two types of responses. An EEG/MEG analysis might also focus on the differences in time courses between the responses to the two types of images.

Thus, the starting point for many analyses is the association of labels corresponding to different **experimental conditions** with time points in the data recording. In BIDS, this association is stored an `events.tsv` file paired with the data recording, but this information may also be stored as part of the recording itself, depending on the technology and the format of the recording.

6.9.2.1 Design matrices and factor variables

The type of information included for the experimental conditions and how this information is stored depends very much on the experiment. Most analysis tools require a vector (sometimes called a **factor vector**) of elements associated with the event markers for each type of experimental condition.

For linear modeling and other types of regression, these factor vectors are assembled into **design matrix** to use as input for the analysis. Design matrices can also include other types of columns depending on the modeling strategy.

6.9.2.2 Types of condition encoding

Consider the simple example introduced above of an experiment which varies the stimuli between pictures of houses and faces to measure differences in response. The following example shows three possible types of encodings (**categorical**, **ordinal**, and **one-hot**) that might be used for this association. The table shows an excerpt from a putative events file, with the onset column (required by BIDS) containing the time of the event marker relative to the start of the associated data recording. The duration column (also required by BIDS) contains the duration of the image presentation in seconds.

Example 14: Illustration of categorical and one-hot encoding of categorical variables.

onset	duration	categorical	ordinal	one_hot.house	one_hot.face
2.010	0.1	house	1	1	0
3.210	0.1	house	1	1	0
4.630	0.1	face	2	0	1
6.012	0.1	house	1	1	0
7.440	0.1	face	2	0	1

The **categorical** encoding assigns laboratory-specific names to the different types of stimuli. In theory, this categorical column consisting of the strings *house* and *face* could be used as a factor vector or as part of a design matrix for regression. However, many analysis tools require that these names be assigned numerical values.

The **ordinal** encoding assigns an arbitrary sequence of numbers corresponding to the unique values. If there are only 2 values, the values -1 and 1 are often used. Ordinal encodings impose an order based on the values chosen, which may have undesirable affects on the results of analyses such as regression if the ordering/relative sizes do not reflect the properties of the encoded experimental conditions.

In Example 14, the experimental conditions houses and faces do not have an ordering/size relationship reflected by the encoding (house=1, face=2). In addition, neither categorical nor ordinal encoding can represent items falling into multiple categories of the same condition at the same time. For these reasons, many statistical tools require one-hot encoding.

In **one-hot** encoding, each possible value of the condition is represented by its own column with 1's representing the presence of that condition value and experimental conditions and 0's otherwise. One-hot encodes all values without bias and allows for a given event to be a member of multiple categories. This representation is required for many machine-learning models. A disadvantage is that it can generate a large number of columns if there are many unique categorical values. It can also cause a problem if not all files contain the same values, as then different files may have different columns.

6.10 File remodeling quickstart

This tutorial works through the process of restructuring tabular (`.tsv`) files using the HED file remodeling tools. These tools particularly useful for creating event files from information in experimental logs and for restructuring event files to enable a particular analysis.

The tools, which are written in Python, are designed to be run on an entire dataset. This dataset can be in BIDS (**Brain Imaging Data Structure**), Alternative users can specify files with a particular suffix and extension appearing in a specified directory tree. The later format is useful for restructuring that occurs early in the experimental process, for example, during the conversion from the experimental control software formats.

The tools can be run using a command-line script, called from a Jupyter notebook, or run using online tools. This quickstart covers the basic concepts of remodeling and develops some basic examples of how remodeling is used. See the ***File remodeling tools*** guide for detailed descriptions of the available operations.

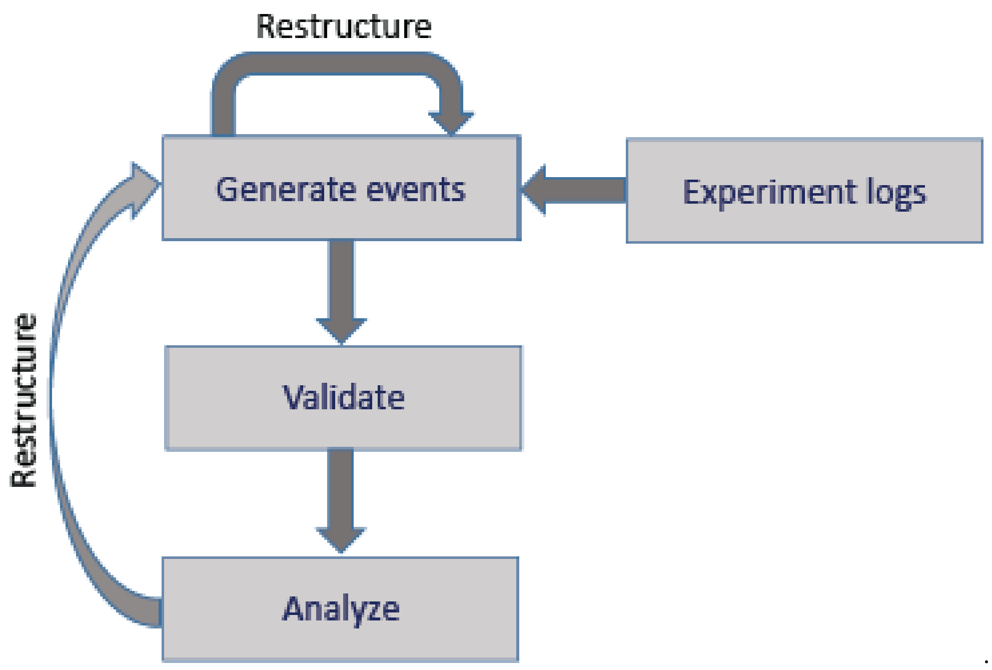
- ***What is remodeling?***
- ***The remodeling process***
- ***JSON remodeling files***
  - ***Basic remodel operation syntax***
  - ***Applying multiple remodel operations***
  - ***More complex remodeling***
  - ***Remodeling file locations***
- ***Using the remodeling tools***
  - ***Online tools for debugging***
  - ***The command-line interface***
  - ***Jupyter notebooks for remodeling***

6.10.1 What is remodeling?

Although the remodeling process can be applied to any tabular file, they are most often used for restructuring event files. Event files, which consist of identified time markers linked to the timeline of the experiment, provide a crucial bridge between what happens in the experiment and the experimental data.

Event files are often initially created using information in the log files generated by the experiment control software. The entries in the log files mark time points within the experimental record at which something changes or happens (such as the onset or offset of a stimulus or a participant response). These event files are then used to identify portions of the data corresponding to particular points or blocks of data to be analyzed or compared.

Remodeling refers to the process of file restructuring including creating, modifying, and reorganizing tabular files in order to disambiguate or clarify their information to enable or streamline their analysis and/or further distribution. HED-based remodeling can occur at several stages during the acquisition and processing of experimental data as shown in this schematic diagram:



In addition to restructuring during initial structuring of the tabular files, further event file restructuring may be useful when the event files are not suited to the requirements of a particular analysis. Thus, restructuring can be an iterative process, which is supported by the HED Remodeling Tools for datasets with tabular event files.

The following table gives a summary of the tools available in the HED remodeling toolbox.

Table 2: Summary of the HED remodeling operations for tabular files.

Category	Operation	Example use case
clean-up		
	remove_columns	Remove temporary columns created during restructuring.
	remove_rows	Remove rows with a particular value in a specified column.
	rename_columns	Make columns names consistent across a dataset.
	reorder_columns	Make column order consistent across a dataset.
factor		
	factor_column	Extract factor vectors from a column of condition variables.
	factor_hed_tags	Extract factor vectors from search queries of HED annotations.
	factor_hed_type	Extract design matrices and/or condition variables.
restructure		
	merge_consecutive	Replace multiple consecutive events of the same typewith one event of longer duration.
	remap_columns	Create *m* columns from values in *n* columns (for recoding).
	split_rows	Split trial-encoded rows into multiple events.
summarization		
	summarize_column_names	Summarize column names and order in the files.
	summarize_column_values	Count the occurrences of the unique column values.
	summarize_definitions	Summarize definitions used and report inconsistencies.
	summarize_hed_tags	Summarize the HED tags present in the HED annotations for the dataset.
	summarize_hed_type	Summarize the detailed usage of a particular type tag such as *Condition-variable* or *Task* (used to automatically extract experimental designs).
	summarize_hed_validation	Validate the data files and report any errors.
	summarize_sidecar_from_events	Generate a sidecar template from an event file.

The **clean-up** operations are used at various phases of restructuring to assure consistency across files in the dataset.

The **factor** operations produce column vectors of the same length as the number of rows in a file in order to encode condition variables, design matrices, or the results of other search criteria. See the ***HED conditions and design matrices*** for more information on factoring and analysis.

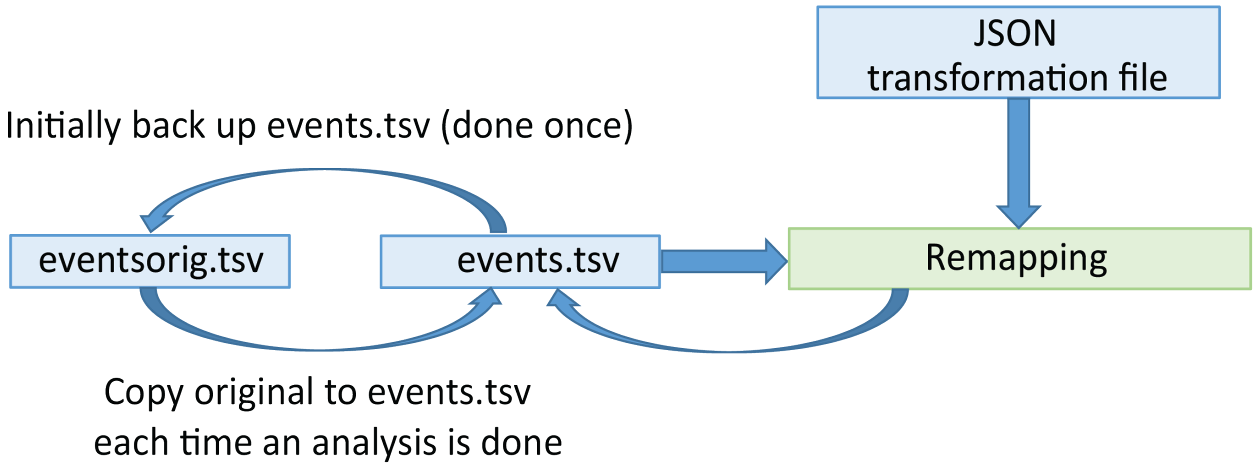
The **restructure** operations modify the way that files represent information.

The **summarization** operations produce dataset-wide and individual file summaries of various aspects of the data.

More detailed information about the remodeling operations can be found in the ***File remodeling tools*** guide.

6.10.2 The remodeling process

Remodeling consists of applying a list of operations to a tabular file to restructure or modify the file in some way. The following diagram shows a schematic of the remodeling process.


JSON transformation file
Initially back up events.tsv (done once)
eventsorig.tsv
events.tsv
Remapping
Copy original to events.tsv each time an analysis is done

Initially, the user creates a backup of the selected files. This backup process is performed only once, and the results are stored in the `derivatives/remodel/backups` subdirectory of the dataset.

Restructuring applies a sequence of remodeling operations given in a JSON remodeling file to produce a final result. By convention, we name these remodeling instruction files `_rmdl.json` and store them in the `derivatives/remodel/remodeling_files` directory relative to the dataset root directory.

The restructuring always proceeds by looking up each data file in the backup and applying the transformation to the backup before overwriting the non-backed up version.

The remodeling file provides a record of the operations performed on the file starting with the original file. If the user detects a mistake in the transformation instructions, he/she can correct the remodeling JSON file and rerun.

Usually, users will use the default backup, run the backup request once, and work from the original backup. However, user may also elect to create a named backup, use the backup as a checkpoint mechanism, and develop scripts that use the check-pointed versions as the starting point. This is useful if different versions of the events files are needed for different purposes.

6.10.3 JSON remodeling files

The operations to restructure a tabular file are stored in a remodel file in JSON format. The file consists of a list of JSON dictionaries.

6.10.3.1 Basic remodel operation syntax

Each dictionary specifies an operation, a description of the purpose, and the operation parameters. The basic syntax of a remodeler operation is illustrated in the following example which renames the *trial_type* column to *event_type*.

Example of a remodeler operation.

```
{
    "operation": "rename_columns",
    "description": "Rename a trial type column to more specific event_type",
    "parameters": {
        "column_mapping": {
            "trial_type": "event_type"
        },
        "ignore_missing": true
    }
}
```

Each remodeler operation has its own specific set of required parameters that can be found under ***File remodeling tools***. For *rename_columns*, the required operations are *column_mapping* and *ignore_missing*. Some operations also have optional parameters.

6.10.3.2 Applying multiple remodel operations

A remodel JSON file consists of a list of one or remodel operations, each specified in a dictionary. These operations are performed by the remodeler in the order they appear in the file. In the example below, a summary is performed after renaming, so the result reflects the new column names.

An example JSON remodeler file with multiple operations.

```
[
    {
        "operation": "rename_columns",
        "description": "Rename a trial type column to more specific event_type.",
        "parameters": {
            "column_mapping": {
                "trial_type": "event_type"
            },
            "ignore_missing": true
        }
    },
    {
        "operation": "summarize_column_names",
        "description": "Get column names across files to find any missing columns.",
        "parameters": {
            "summary_name": "Columns after remodeling",
            "summary_filename": "columns_after_remodel"
        }
    }
]
```

By stacking operations you can make several changes to a data file, which is important because the changes are always applied to a copy of the original backup. If you are planning new changes to the file, note that you are always changing a copy of the original backed up file, not a previously remodeled `.tsv`.

6.10.3.3 More complex remodeling

This section discusses a complex example using the **sub-0013_task-stopsignal_acq-seq_events.tsv** events file of AOMIC-PIOP2 dataset available on OpenNeuro as ds002790. Here is an excerpt of the event file.

Excerpt from an event file from the stop-go task of AOMIC-PIOP2 (ds002790).

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex
0.0776	0.5083	go	n/a	0.565		correct	right
5.5774	0.5083	unsuccesful_stop	0.2	0.49	correct	right	female
9.5856	0.5084	go	n/a	0.45	correct	right	female
13.5939	0.5083	succesful_stop	0.2	n/a	n/a	n/a	female
17.1021	0.5083	unsuccesful_stop	0.25	0.633	correct	left	male
21.6103	0.5083	go	n/a	0.443	correct	left	male

This event file corresponds to a stop-signal experiment. Participants were presented with faces and had to decide the sex of the face by pressing a button with left or right hand. However, if a stop signal occurred before this selection, the participant was to refrain from responding.

The structure of this file corresponds to the **BIDS** format for event files. The first column, which must be called `onset` represents the time from the start of recording in seconds of the temporal marker represented by that row in the file. In this case that temporal marker represents the presentation of a face image.

Notice that the *stop_signal_delay* and *response_time* columns contain information about additional events (when a trial stop signal was presented and when the participant pushed a button). These events are encoded implicitly as offsets from the presentation of the go signal. Each row is the file encodes information for an entire trial rather than what occurred at a single temporal marker. This strategy is known as *trial-level* encoding.

Our goal is to represent all the trial events (e.g., go signal, stop signal, and response) in separate rows of the event file using the *split_rows* restructuring operation. The following example shows the remodeling operations to perform the splitting.

Example of split_rows operation for the AOMIC stop signal task.

```
[
    {
        "operation": "split_rows",
        "description": "Split response event from trial event based on response_time column.",
        "parameters": {
            "anchor_column": "trial_type",
```

(continues on next page)

(continued from previous page)

```
            "new_events": {
                "response": {
                    "onset_source": ["response_time"],
                    "duration": [0],
                    "copy_columns": ["response_accuracy", "response_hand"]
                },
                "stop_signal": {
                    "onset_source": ["stop_signal_delay"],
                    "duration": [0.5],
                    "copy_columns": []
                }
            },
            "remove_parent_row": false
        }
    }
]
```

The example uses the *split_rows* operation to convert this file from trial encoding to event encoding. In trial encoding each event marker (row in the event file) represents all the information in a single trial. Event markers such as the participant's response key-press are encoded implicitly as an offset from the stimulus presentation. while event encoding includes event markers for each individual event within the trial.

The ***Split rows*** explanation under ***File remodeling tools*** shows the required parameters for the *split_rows* operation. The required parameters are *anchor_column*, *new_events*, and *remove_parent_row*.

The *anchor_column* is the column we want to add new events corresponding to the stop signal and the response. In this case we are going to add events to an existing column: *trial_type*. The new events will be in new rows and the existing rows will not be overwritten because *remove_parent_event* is false. (After splitting we may want to rename *trial_type* to *event_type* since the individual rows in the data file no longer represent trials, but individual events within the trial.)

Next we specify how the new events are generated in the *new_events* dictionary. Each new event has a name, which is a key in the *new_events* dictionary. For each key is associated with a dictionary specifying the values of the following parameters.

- *onset_source*
- *duration*
- *copy_columns*`

The *onset_source* is a list indicating how to calculate the onset for the new event relative to the onset of the anchor event. The list contains any combination of column names and numerical values, which are evaluated and added to the onset value of the row being split. Column names are evaluated to the row values in the corresponding columns.

In our example, the response time and stop signal delay are calculated relative to the trial's onset, so we only need to add the value from the respective column. Note that these new events do not exist for every trial. Rows where there was no stop signal have an *n/a* in the *stop_signal_delay* column. This is processed automatically, and remodeler does not create new events when any items in the *onset_source* list is missing or *n/a*.

The *duration* specifies the duration for the new events. The AOMIC data did not measure the durations of the button presses, so we set the duration of the response event to 0. The AOMIC data report indicates that the stop signal lasted 500 ms.

The copy columns indicate which columns from the parent event should be copied to the newly-created event. We would like to transfer the *response_accuracy* and the *response_hand* information to the response event.

The final remodeling file can be found at: **finished json remodeler**

6.10.3.4 Remodeling file locations

The remodeling tools expect the full path for the JSON remodeling operation file to be given when the remodeling is executed. However, it is a good practice to include all remodeling files used with the dataset. The JSON remodeling operation files are usually located in the `derivatives/remodel/remodeling_files` subdirectory below the dataset root, and have file names ending in `_rmdl.json`.

The backups are always in the `derivatives/remodel/backups` subdirectory under the dataset root. Summaries produced by the restructuring tools are located in `derivatives/remodel/summaries`.

In the next section we will go over several ways to call the remodeler.

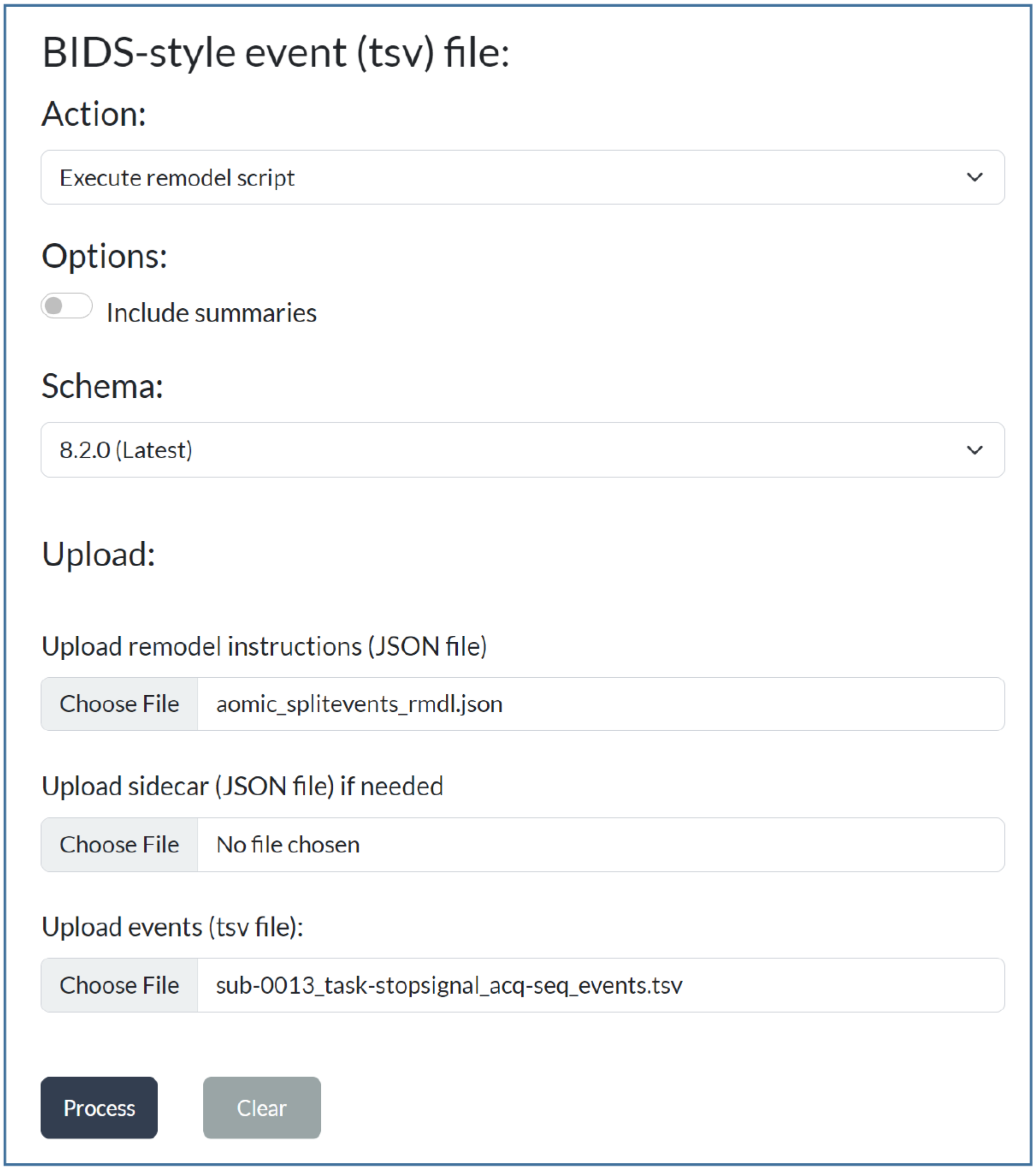
6.10.4 Using the remodeling tools

The remodeler can be called in a number of ways including using online tools and from the command line. The following sections explain various ways to use the available tools.

6.10.4.1 Online tools for debugging

Although the event restructuring tools are designed to be run on an entire dataset, you should consider working with a single data file during debugging. The HED online tools provide support for debugging your remodeling script and for seeing the effect of remodeling on a single data file before running on the entire dataset. You can access these tools on the **HED tools online tools server**.

To use the online remodeling tools, navigate to the events page and select the *Execute remodel script* action. Browse to select the data file to be remodeled and the JSON remodel file containing the remodeling operations. The following screenshot shows these selections for the split rows example of the previous section.



BIDS-style event (tsv) file:

Action:

Execute remodel script

Options:

Include summaries

Schema:

8.2.0 (Latest)

Upload:

Upload remodel instructions (JSON file)

Choose File aomic_splitevents_rmdl.json

Upload sidecar (JSON file) if needed

Choose File No file chosen

Upload events (tsv file):

Choose File sub-0013_task-stopsignal_acq-seq_events.tsv

Press the *Process* button to complete the action. If the remodeling script has errors, the result will be a downloaded text file with the errors identified. If the remodeling script is correct, the result will be a data file with the remodeling transformations applied. If the remodeling script contains summarization operations, the result will be a zip file with the modified data file and the summaries included.

If you are using one of the remodeling operations that relies on HED tags, you will also need to upload a suitable JSON sidecar file containing the HED annotations for the data file if you turn the *Include summaries* option on.

6.10.4.2 The command-line interface

After *installing the remodeler*, you can run the tools on a full BIDS dataset, or on any directory using the command-line interface using `run_remodel_backup`, `run_remodel`, and `run_remodel_restore`. A full overview of all arguments is available at *File remodeling tools*.

The `run_remodel_backup` is usually run only once for a dataset. It makes the baseline backup of the event files to assure that nothing will be lost. The remodeling always starts from the backup files.

The `run_remodel` restores the data files from the corresponding backup files and then executes remodeling operations from a JSON file. A sample command line call for `run_remodel` is shown in the following example.

Command to run a summary for the AOMIC dataset.

```
python run_remodel /data/ds002790  /data/ds002790/derivatives/remodel/remodeling_files/
↪AOMIC_summarize_rmdl.json \
 -b -s .txt -x derivatives
```

The parameters are as follows:

- `data_dir` - (Required first argument) Root directory of the dataset.
- `model_path` - (Required second argument) Path of JSON file with remodeling operations.
- `-b` - (Optional) If present, assume BIDS formatted data.
- `-s` - (Optional) list of formats to save summaries in.
- `-x` - (Optional) List of directories to exclude from event processing.

There are three types of command line arguments:

Positional arguments, *Named arguments*, and *Named arguments with values*.

The positional arguments, `data_dir` and `model_path` are not optional and must be the first and second arguments to `run_remodel`. The named arguments (with and without values) are optional. They all have default values if omitted.

The `-b` option is a named argument indicating whether the dataset is in BIDS format. If in BIDS format, the remodeling tools can extract information such as the HED schema and the HED annotations from the dataset. BIDS data file names are unique, which is convenient for reporting summary information. Name arguments are flags– their presence indicates true and absence indicates false.

The `-s` and `-x` options are examples of named arguments with values. The `-s .txt` specifies that summaries should be saved in text format. The `-x derivatives` indicates that the `derivatives` subdirectory should not be processed during remodeling.

This script can be run multiple times without doing backups and restores, since it always starts with the backed up files.

The first argument of the command line scripts is the full path to the root directory of the dataset. The `run_remodel` requires the full path of the json remodeler file as the second argument. A number of optional key-value arguments are also available.

After the `run_remodel` finishes, it overwrites the data files (not the backups) and writes any requested summaries in `derivatives/remodel/summaries`.

The summaries will be written to `/data/ds002790/derivatives/remodel/summaries` folder in text format. By default, the summary operations will return both.

The **summary file** lists all different column combinations and for each combination, the files with those columns. Looking at the different column combinations you can see there are three, one for each task that was performed for this dataset.

Going back to the *split rows example* of remodeling, we see that splitting the rows into multiple rows only makes sense if the event files have the same columns. Only the event files for the stop signal task contain the `stop_signal_delay` column and the `response_time` column. The summarizing the column names across the dataset allows users to check whether the column names are consistent across the dataset. A common use case for BIDS datasets is that the event files have a different structure for different tasks.
The `-t` command-line option allows users to specify which tasks to perform remodeling on. Using this option allows users to select only the files that have the specified task names in their filenames.

Now you can try out the *split_rows* on the full dataset!

6.10.4.3 Jupyter notebooks for remodeling

Three Jupyter remodeling notebooks are available at **Jupyter notebooks for remodeling**.

These notebooks are wrappers that create the backup as well as run restructuring operations on data files. If you do not have access to a Jupyter notebook facility, the article Six easy ways to run your Jupyter Notebook in the cloud discusses various no-cost options for running Jupyter notebooks online.

6.11 HED schema developer's guide

HED annotations consist of comma-separated terms drawn from a hierarchically structured vocabulary called a HED schema. The **HED standard schema** contains basic terms that are common across most human neuroimaging, behavioral, and physiological experiments. The HED ecosystem also supports (**HED library schemas**) to expand the HED vocabulary in a scalable manner to support specialized data.

Although you can create a private HED vocabulary for your

This guide describes how to begin developing your own schema.

This section describes how you can contribute to existing HED vocabularies or creating an entirely new one.

6.11.1 Setting up for schema development

Although schema developers work with HED schema in `.mediawiki` format for ease in editing, HED tools generally use XML versions of the HED schema.

Standard development process for XML schema.

1. Create or modify a `.mediawiki` file containing the schema.
2. Validate the `.mediawiki` file using the **HED online tools**.
3. Convert to `.xml` using the **HED online tools**.
4. View in the **expandable schema viewer** to verify.

6.11.2 Design principles for schema

All HED schema (both the standard and library schemas) must conform to certain design principles in addition to properly validating.

Rules for HED schema design.

1. [**Unique**] Every term must be unique within the schema and must conform to the rules for HED schema terms.
2. [**Meaningful**] Schema terms should be readily understood by most users. The terms should not be ambiguous and should be meaningful in themselves **without** reference to their position in the schema hierarchy.
3. [**Organized**] If possible, a schema sub-tree should have no more than 7 direct subordinate sub-trees.
4. [**Orthogonal**] Terms that are used independently of one another should be in different sub-trees (orthogonality).
5. [**Sub-classed**]Every term in the hierarchy satistifies the **is-a** relationship with its parent. In other words if B has A as a parent in the schema hierarchy, then B is an example of A. Searching for A will also return B (search generality).

As in Python programming, we anticipate that many HED schema libraries may be defined and used, in addition to the base HED schema. Libraries allow individual research or clinical communities to annotate details of events in experiments designed to answer questions of interest to particular to those communities.

Since it would be impossible to avoid naming conflicts across schema libraries that may be built in parallel by different user communities, HED supports schema library namespaces. Users will be able to add library tags qualified with namespace designators. All HED schemas, including library schemas, adhere to semantic versioning.

6.11.3 Defining a schema

A HED library schema is defined in the same way as the base HED schema except that it has an additional attribute name-value pair, `library="xxx"` in the schema header. We will use as an illustration a library schema for driving. Syntax details for a library schema are similar to those for the base HED schema. (See the HED schema format specification for more details).

Example: Driving library schema (MEDIAWIKI template).

```
HED library="driving" version="1.0.0"
!# start schema
   [... contents of the HED driving schema ...]
!# end schema
   [... required sections specifying schema attribute definitions ...]
!# end hed
```

The required sections specifying the schema attributes are *unit-class-specification*, *unit-modifier-specification*, *value-class-specification*, *schema-attribute-specification*, and *property-specification*.

Example: Driving library schema (XML template).

```
<?xml version="1.0" ?>
<HED library="driving" version="1.0.0">
    [... contents of the HED_DRIVE schema ... ]
</HED>
```

The schema XML file should be saved as `HED_driving_1.0.0.xml` to facilitate specification in tools.

6.11.4 Schema namespaces

As part of the HED annotation process, users must associate a standard HED schema with their datasets. Users may also include tags from an arbitrary number of additional library schemas. For each library schema used to annotate a data recording, the user must associate a local name with the appropriate library schema name and version. Each library must be associated with a distinct local name within a recording annotations. The local names should be strictly alphabetic with no blanks or punctuation.

The user must pass information about the library schema and their associated local names to processing functions. HED uses a standard method of identifying namespace elements by prefixing HED library schema tags with the associated local names. Tags from different library schemas can be intermixed with those of the base schema. Since the node names within a library must be unique, annotators can use short form as well as fully expanded tag paths for library schema tags as well as those from the base-schema.

Example: Driving library schema example tags.

```
dp:Action/Drive/Change-lanes
dp:Drive/Change-lanes
dp:Change-lanes
```

A colon (`:`) is used to separate the qualifying local name from the remainder of the tag. Notice that *Action* also appears in the standard HED schema. Identical terms may be used in a library schema and the standard HED schema. Use of the same term implies a similar purpose. Library schema developers should try not to reuse terms in the standard schema unless the intention is to convey a close or identical relationship.

6.11.5 Attributes and classes

In addition to the specification of tags in the main part of a schema, a HED schema has sections that specify unit classes, unit modifiers, value classes, schema attributes, and properties. The rules for the handling of these sections for a library schema are as follows:

6.11.5.1 Required sections

The required sections of a library schema are: the *schema-specification*, the *unit-class-specification*, the *unit-modifier-specification*, the *value-class-specification* section, the *schema-attribute-specification* section, and the *property-specification*. The library schema must include all required schema sections even if the content of these sections is empty.

6.11.5.2 Relation to base schema

Any schema attribute, unit class, unit modifier, value class, or property used in the library schema must be specified in the appropriate section of the library schema regardless of whether these appear in base schema. Validators check the library schema strictly on the basis of its own specification without reference to another schema.

6.11.5.3 Schema properties

HED only supports the schema properties listed in Table B.2: *boolProperty*, *unitClassProperty*, *unitModifierProperty*, *unitProperty*, and *valueClassProperty*.
If the library schema uses one of these in the library schema specification, then its specification must appear in the *property-specification* section of the library schema.

6.11.5.4 Unit classes

The library schema may define unit classes and units as desired or include unit classes or units from the base schema. Similarly, library schema may define unit modifiers or reuse unit modifiers from the base schema. HED validation and basic analysis tools validate these based strictly on the schema specification and do not use any outside information for these.

6.11.5.5 Value classes

The standard value classes (*dateTimeClass[*], nameClass, numericClass[*]*, *posixPath[*], textClass[*]*) if used, should have the same meaning as in the base schema. The hard-coded behavior associated with the starred ([*]) value classes will be the same. Library schema may define additional value classes and specify their allowed characters, but no additional hard-coded behavior will be available in the standard toolset. This does not preclude special-purpose tools from incorporating their own behavior.

6.11.5.6 Schema attributes

The standard schema attributes (*allowedCharacter*, *defaultUnits*, *extensionAllowed*, *recommended*, *relatedTag*, *requireChild*, *required*, *SIUnit*, *SIUnitModifier*, *SIUnitSymbolModifier*, *suggestedTag*, *tagGroup*, *takesValue*, *topLevelTagGroup*, *unique*, *unitClass*, *unitPrefix*, *unitSymbol*, *valueClass*) should have the same meaning as in the base schema. The hard-coded behavior associated with the schema attributes will be the same. Library schema may define additional schema attributes. They will be checked for syntax, but no additional hard-coded behavior will be available in the standard toolset. This does not preclude special-purpose tools from incorporating their own behavior.

6.11.5.7 Syntax checking

Regardless of whether a specification is in the standard schema or a library schema, HED tools can perform basic syntax checking.

Basic syntax checking for library schema.

1. All attributes used in the schema proper must be defined in the schema attribute section of the schema.
2. Undefined attributes cause an error in schema validation.
3. Similar rules apply to unit classes, unit modifiers, value classes, and properties.
4. Actual handling of the semantics by HED tools only occurs for entities appearing in the base schema.

6.11.5.8 Procedure for updating a schema.

Proposing changes

As modifications to the HED schema are proposed, they are added to the **PROPOSED.md** file for the respective schema. As changes are accepted, they are incorporated into the **prerelease** version of the schema and added as part of the **prerelease CHANGES.md**. These files are located in the **prerelease** subdirectory for the respective schema. Examples of these files for the standard schema can be found in the standard schema **prerelease directory**. **Expandable html view of the prerelease HED schema**

Upon final review, the new HED schema is released, the XML file is copied to the **hedxml directory**, the mediawiki file is copied to the [**hed]

6.11.6 HED schema details

HED schema is the structured vocabulary from which HED annotations base on. HED annotations consist of comma-separated path strings, selected from the schema. In the newest versions of HED, all individual nodes in the vocabulary are unique, so users can annotate by simply giving the last node in the path string rather than the entire path string: *Red* instead of *Attribute/Sensory/Sensory-property/Visual/Color/CSS-color/Red-color/Red*.

This repository contains the HED schema specification, where discussions on schema terms and syntax are held via Github issue mechanism and where HED-supporting tools can find machine-readable format of the schema. The HED schema is available in MediaWiki and XML.

The MediaWiki markdown format, stored in `hedwiki`, allows vocabulary developers to view and edit the vocabulary tree using a human-readable markdown language available in Wikis and on GitHub repositories. In addition, an expandable non-editable HTML viewer is available to help users explore the vocabulary.

All analysis and validation tools operate on an XML translation of the vocabulary markdown document, stored in `hedxml`.

6.11.7 Further documentation

The documentation on this page refers specifically to the HED vocabulary and supporting tools. Additional documentation is available on:

> **HED organization website**

All of the HED software is open-source and organized into various repositories on the HED standards organization website:

> **HED organization github repository**

6.12 HED online tools

HED web-based tools are available directly through a browser from https://hedtools.ucsd.edu/hed or as RESTful services from the same URL. These services do not require a login to use.

The tools are implemented in a Docker module and can be deployed locally provided that Docker is installed. See the HED Web documentation about download and deployment information.

- ***Browser-based tools*** - web-based tools for HED.
- ***RESTful services*** - RESTful online HED services.

6.12.1 Browser-based access

The HED browser-based tools are organized into the following pages, each focused on a particular type of file.

- *Event online tools* - validation, summary, and generation tools.
- *Sidecar online tools* - validation, transformation, extraction, and merging tools.
- *Spreadsheet online tools* - validation and transformation tools.
- *String online tools* - validation and transformation tools.
- *Schema online tools* - validation and conversion tools.

Many of the tools require that you provide a HED schema. Usually, you can do this by selecting one of the standard HED versions using a pull-down menu, and the tool downloads this version from GitHub if the server doesn't already have it cached.

If you want to use a different version of HED, you can select the *Other* option from the pull-down and upload your own HED schema.

The long form HED tag consists of the tag's full path in the HED schema, while the short form consists only of the tag's leaf node in the schema and possibly a value. Intermediate form tags consist of a partial paths from a leaf node to an intermediate node in the HED schema. Compliant HED tools should be able to handle any combination of short, long, or intermediate form tags.

Several of the tools have an `Expand defs` option to indicate that definitions should be expanded. When this option is in effect, tools should replace *Def/xxx* tags with an expanded definition for 'xxx' with a tag group of the form *(Def-expand/xxx, yyy)* where 'yyy' is the actual definition contents of 'xxx'.

Note: Expansion of definitions is independent of whether the individual HED tags are in long form or short form.

6.12.1.1 Events files

Events files are BIDS style tab-separated value files. The first line is always a header line giving the names of the columns, which are used as keys to metadata in accompanying JSON sidecars.

The HED tools have four separate tools: validate, assemble annotations, generate sidecar template, and execute remodel script.

Validate an events file

The validate tool for events is useful for debugging the HED annotations in your BIDS dataset while avoiding a full BIDS-validation each time you make a change. The tool first validates the sidecar if present and then does a final validation in combination with the events file.

Validate a BIDS-style events file

Steps:

- Select the `Validate` action.
- Set `Check for warnings` on if you want to include warnings.
- Select the HED version.
- Optionally upload a JSON sidecar file (`.json`).
- Upload an events file (`.tsv`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages.

The online events file validation tool is very useful for quick validation while developing your annotation. However, the tool only validates a single events file with an accompanying sidecar. The tool does not validate multiple events files at the same time, nor does the tool handle inherited sidecars.

The bids_validate_hed.ipynb Python Jupyter notebook is available for validating all the events files in a BIDS dataset along with multiple sidecars. The Jupyter notebook handles validation with library schema.

Assemble annotations

Assembling HED annotations of a BIDS-style events file produces a two-column result file whose first column contains the onsets of the original events file and the second column contains the fully assembled HED annotation for each event.

Assemble HED annotations for a BIDS-style events file.

Steps:

- Select the `Assemble annotations` action.
- Set `Expand defs` on if you want to include expanded definitions.
- Specify the HED version.
- Optionally upload a JSON sidecar file (`.json`).
- Upload the events file (`.tsv`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns the assembled `.tsv` events file.

The online tools do not allow the option of retaining other columns in the returned file. A more general alternative is to use the remodeling tools through the interface.

Generate sidecar template

Generating a sidecar template file from the information in a single events file produces a `.json` sidecar template file ready to be filled in with descriptions and HED annotations.

Generate a sidecar template from an events file.

Steps:

- Select the `Generate sidecar template` action.
- Upload the events file (`.tsv`). A list of the event file column names with number of unique values in each column appears below the selection.
- In the left column of checkboxes select those corresponding to columns you wish to include in the template.
- In the right column of checkboxes, select those corresponding to columns for which you wish to supply individual annotations for each unique value.

- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable `.json` sidecar template file corresponding to the events file.

The online generation tool is very useful for constructing a sidecar template file, but the template is based only on the particular events file used in the generation. For many BIDS datasets, this is sufficient for generating a complete template. However, for datasets that have many types of events files, you will want to use the bids_generate_sidecar.ipynb to generate a JSON sidecar based on all the events files in a BIDS dataset.

Execute remodel script

The HED remodeling tools provide an interface to nearly all the HED tools functionality without programming. To use the tools, create a JSON file containing the commands that you wish to execute on the events file. Command are available to do various transformations and summaries of events files as explained in the **File remodeling quickstart** and the **File remodeling tools**.

Execute a remodel script.

Steps:

- Select the `Execute remodel script` action.
- Set `Include summaries` if you wish to get the summary output.
- Specify the HED version.
- Upload a JSON file containing the remodel commands.
- Optionally upload a JSON sidecar file (`.json`).
- Upload the events file (`.tsv`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages. If there are no errors, the tool returns a zip archive containing the transformed events file and any possible summaries that were generated.

6.12.1.2 Sidecar files

BIDS JSON sidecars have file names ending in `events.json`. These JSON files contain metadata and HED tags applicable to associated events files.

Validate a sidecar

The validate tool for sidecars is useful for debugging the HED annotations in your BIDS dataset while avoiding a full BIDS-validation each time you make a change. The tool validates a single JSON sidecar.

Validate a BIDS style JSON sidecar.

- Select the `Validate` action.
- Set `Check for warnings` to on if you want to include warnings.
- Select the HED version.
- Upload a JSON sidecar file (`.json`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages.

The online validation tool is very useful for quick validation while developing your annotation. For datasets that have all of their HED annotations in a single JSON sidecar in the dataset root directory, this is all that is needed.

However, if the sidecar is part of an inheritance chain, some of its definitions are externally defined, or the sidecar contains tags from multiple HED schemas, you should use the bids_validate_dataset.ipynb Python Jupyter notebook to validate the HED in your BIDS dataset.

Convert sidecar to long

The convert sidecar to long tool first does a preliminary validation of the sidecar to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

If successful, the convert sidecar to long tool produces a new sidecar file with all the HED tags in full long-form. The non-HED portions of the sidecar are the same as in the original file.

Convert a JSON sidecar HED tags to long form.

Steps:

- Select the `Convert to long` action.
- Set `Expand defs` to on if you want to include expanded definitions.
- Specify the HED version.
- Upload the JSON sidecar file (`.json`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable converted JSON sidecar file.

Convert sidecar to short

The convert sidecar to short tool first does a preliminary validation of the sidecar to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

If successful, the convert sidecar to short tool produces a new sidecar file with all the HED tags in short form, making the sidecar easier to read and work with. The non-HED portions of the sidecar are the same as in the original file.

Convert a JSON sidecar HED tags to short form.

Steps:

- Select the `Convert to short` action.
- Set `Expand defs` to on if you want to include expanded definitions.
- Specify the HED version.
- Upload the JSON sidecar file (`.json`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable converted JSON sidecar file.

Extract spreadsheet from sidecar

JSON sidecars are sometimes hard to edit, particularly if the annotations are complicated. The extract spreadsheet from sidecar tool produces a 4-column `.tsv` file that can be edited with tools such as Excel. The first row of the extracted spreadsheet contains the 4 column names: `column_name`, `column_value`, `description` and `HED`. See the **BIDS annotation quickstart** for a tutorial on how to use the resulting spreadsheet for annotation.

Extract a 4-column spreadsheet from a JSON sidecar.

Steps:

- Select the `Extract HED spreadsheet` action.
- Upload the JSON sidecar file (`.json`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable `.tsv` spreadsheet.

The bids_sidecar_to_spreadsheet.ipynb Python Jupyter notebook does the same operation.

Merge a spreadsheet with a sidecar

This tool merges a 4-column tag spreadsheet with an existing JSON file to produce a JSON file that contains HED annotations updated with the information from the spreadsheet. The spreadsheet can be in either tab-separated (`.tsv`) or in Excel (`.xlsx`) format, but it must have the 4 column names: `column_name`, `column_value`, `description` and `HED`.

You have the option of including the contents of each cell in the *description* column of the spreadsheet as a *Description/xxx* tag in the corresponding HED annotation.

See the **BIDS annotation quickstart** for a tutorial on how this works in practice.

Merge a 4-column spreadsheet with a JSON sidecar.

Steps:

- Select the `Merge HED spreadsheet` action.
- Set `Include Description tags` to on if you want to include descriptions.
- Upload the target JSON sidecar file (`.json`).
- Upload the spreadsheet to be merged (`.tsv` or `.xlsx`).
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable merged `.json` file.

The bids_merge_sidecar.ipynb Python Jupyter notebook does the same operation.

6.12.1.3 Spreadsheet files

Spreadsheets (either in Excel or tab-separated-value format) are convenient for organizing tags. Of particular interest is the 4-column spreadsheet described in the **BIDS annotation quickstart**. However, the online tools support a more general spreadsheet format, where any columns can contain HED tags. You can also specify prefixed columns — columns in which a particular column has its values prefixed by a particular HED tag prior to processing. Often prefixing is used for the *Description* tag.

These spreadsheets are not necessarily associated with particular datasets or events files. Rather, they are useful when you are developing annotations in general.

Validate a spreadsheet

The validate tool for spreadsheets is useful for debugging HED annotations while you are developing them. The tool validates a single spreadsheet worksheet, either in tab-separated value (`.tsv`) or Excel (`.xlsx`) format. The Excel format supports spreadsheets containing multiple worksheets, but you must validate each worksheet individually. When you select a spreadsheet or change the individual worksheet being considered, a list of column names will appear with checkboxes on the left and text boxes on the right. Select the checkboxes for columns you wish to validate. Add a prefix tag in the corresponding text box on the left, if the entry is to be prefixed by a particular tag before validating.

Validate a spreadsheet.

- Select the `Validate` action.

- Set `Check for warnings` to on if you want to include warnings.
- Select the HED version.
- Upload a spreadsheet file (`.tsv` or `.xlsx`).
- Select a worksheet if necessary.
- Check the columns that contain HED information and should be validated.
- Enter relevant prefixes in the text boxes to the right of the column names.
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages.

Convert spreadsheet to long

The convert spreadsheet to long tool first does a preliminary validation to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

As with other spreadsheet operations, you will have to provide information about which columns of the spreadsheet contain HED tags that should be converted to long by checking the appropriate boxes on the left next to the desired column names. This option does not have text boxes for prefixes.

If successful, the convert spreadsheet to long tool produces a new spreadsheet file with all the HED tags in full long-form. The non-HED portions of the spreadsheet and the prefix-columns are the same as in the original file.

Convert a spreadsheet to long.

- Select the `Convert to long` action.
- Select the HED version.
- Upload a spreadsheet file (`.tsv` or `.xlsx`).
- Select a worksheet if necessary.
- Check the columns that contain HED information and should be validated.
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable spreadsheet with the HED tags converted to long.

Convert spreadsheet to short

The convert spreadsheet to short tool first does a preliminary validation to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

As with other spreadsheet operations, you will have to provide information about which columns of the spreadsheet contain HED tags that should be converted to short by checking the appropriate boxes on the left next to the desired column names. This option does not have text boxes for prefixes.

If successful, the convert spreadsheet to short tool produces a new spreadsheet file with all the HED tags in short form. The non-HED portions of the spreadsheet and the prefix-columns are the same as in the original file.

Convert a spreadsheet to short form.

- Select the `Convert to short` action.
- Select the HED version.
- Upload a spreadsheet file (`.tsv` or `.xlsx`).
- Select a worksheet if necessary.
- Check the columns that contain HED information and should be validated.
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherise the tool returns a downloadable spreadsheet with the HED tags converted to short.

6.12.1.4 String online tools

While in the process of annotating or working with HED, you might find it convenient to do a quick check or conversion of a HED string, particularly when you are building complex annotations. The HED string online tools are useful for this.

Validate a HED string

The validate tool for HED strings validates a single HED string. The HED string may contain multiple HED tags and parenthesized groups of HED tags.

Validate a HED string.

- Select the `Validate` action.
- Set `Check for warnings` to on if you want to include warnings.
- Select the HED version.
- Type or paste your HED string into the text box.
- Click the `Process` button.

Returns: Errors are displayed in the *Results* text box at the bottom of the page.

Convert a HED string to long

The convert string to long tool first does a preliminary validation of the string to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

If successful, the convert to long tool displays the converted string in the *Results* text box at the bottom of the page. You can then use copy or cut with paste to use the converted string in other documents.

Convert HED string to long form.

Steps:

- Select the `Convert to long` action.
- Specify the HED version.
- Type or paste your HED string into the text box.
- Click the `Process` button.

Returns:
If there are any errors, the tool displays the error messages in the *Results* at the bottom of the page, otherwise the tool displays the converted string in the *Results* textbox.

Convert HED string to short

The convert string to short tool first does a preliminary validation of the string to detect errors that prevent conversion from being successful. You should always do a full validation prior to doing conversion.

If successful, the convert to short tool displays the converted string in the *Results* text box at the bottom of the page. You can then use copy or cut with paste to use the converted string in other documents.

Convert HED string to short form.

Steps:

- Select the `Convert to short` action.
- Specify the HED version.
- Type or paste your HED string into the text box.
- Click the `Process` button.

Returns:
If there are any errors, the tool displays the error messages in the *Results* at the bottom of the page, otherwise the tool displays the converted string in the *Results* textbox.

6.12.1.5 Schema online tools

HED schema tools are designed to assist HED schema developers and library schema developers in making sure that their schema has the correct form. The schema tools also provide an easy mechanism for converting between `.xml` and `.mediawiki` schema formats.

You can view standard schema using the expandable **HED vocabulary viewer**.

Validate a HED schema

The validation operation checks syntax as well as HED-3G compliance. Schema nodes must be unique and have a specified format.

Validate a HED schema.

- Select the `Validate` action.
- Set `Check for warnings` on if you want to include warnings.
- Enter a schema URL or upload a schema file (`.xml` or `.mediawiki`) and select the corresponding option.

- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages.

Convert a HED schema

The convert HED schema tool allows you to convert between `.mediawiki` and `.xml` formats. All HED tools use the `.xml` format, but the `.mediawiki` format is much easier to read and modify.

The **HED specification** GitHub repository maintains both versions of the schema.

Convert a HED schema.

- Select the `Convert schema` action.
- Enter a schema URL or upload a schema file (`.xml` or `.mediawiki`) and select corresponding option.
- Click the `Process` button.

Returns:
If there are any errors, the tool returns a downloadable `.txt` file of error messages, otherwise the tool returns a downloadable `.xml` or `.mediawiki` file.

6.12.2 HED RESTful services

HED supports a number of REST web services in support of HED including schema conversion and validation, JSON sidecar validation, spreadsheet validation, and validation of a single BIDS events file with supporting JSON sidecar.

Short-to-long and long-to-short conversion of HED tags are supported for HED strings, JSON sidecars, BIDS-style events files, and spreadsheets in `.tsv` or `.xlsx` format.

Support is also included for assembling the annotations for a BIDS-style events file with a JSON sidecar and for generating a template of a JSON sidecar from a BIDS events file.

The **web_services** directory in the `hed-examples` repository provides MATLAB examples of how to call these services in MATLAB.

6.12.2.1 Service setup

The HED web services are accessed by making a request to the HED web server to obtain a CSRF access token for the session and then making subsequent requests as designed. The steps are:

1. Send an HTTP `get` request to the HED CSRF token access URL.
2. Extract the CSRF token and returned cookie from the response to use in the headers of future `post` requests.
3. Send an HTTP `post` request in the format as described below and read the response.

The following table summarizes the location of the relevant URLs for online deployments of HED web-based tools and services.

Table 3: URLs for HED online services.

Service	URL
Online HED tools	https://hedtools.ucsd.edu/hed
CSRF token access	https://hedtools.ucsd.edu/hed/services
Service request	https://hedtools.ucsd.edu/hed/services_submit

6.12.2.2 Request format

HED services are accessed by passing a JSON dictionary of parameters in a request to the online server. All requests are in JSON format and include a `service` name and additional parameters.

The service names are of the form `target_command` where `target` specifies the input data type (events, sidecar, spreadsheet, string, or schema) and `command`specifies the service to be performed. For example, `events_validate` indicates that a BIDS-style events file is to be validated. The exception to this naming rule is the `get_services` command, which returns a list of all available services and their parameters.

All parameter values are passed as strings. The contents of file parameters are read into strings to be passed as part of the request. The following example shows the JSON for a HED service request to validate a JSON sidecar. The contents of the JSON file to be validated are abbreviated as `"json file text"`.

Example: Request parameters for validating a JSON sidecar.

```
{
    "service": "sidecar_validate",
    "schema_version": "8.0.0",
    "json_string": "json file text",
    "check_for_warnings": "on"
}
```

The parameters are explained in the following table. Parameter values listed in square brackets (e,g, [`a`, `b`]) indicate that only one of `a` or `b`should be provided.

Table 4: Summary of HED ReST services

Service	Parameters	Descriptions
get_services	none	Returns a list of available services.
events_assemble json_string, [schema_version, schema_string], check_for_warnings, expand_defs	events_string, Assemble tags for each event in a BIDS-style events file into a single HED string. Returned data: a file of assembled events as text or an error file as text if errors.	
events_extract	events_string	Extract a template JSON sidecar based on the contents of the events file. Returned data: A JSON sidecar (template) if no errors.
events_validate json_string, [schema_string, schema_url, schema_version], check_for_warnings	events_string, Validate a BIDS-style events file and its JSON sidecar if provided. Returned data: an error file as text if errors.	
sidecar_to_long [schema_string, schema_url, schema_version],	json_string, Convert a JSON sidecar with all of its HED tags expressed in long form. Returned data: a converted JSON sidecar as text or an error file as text if errors.	
sidecar_to_short [schema_string, schema_url, schema_version]	json_string, Convert a JSON sidecar with all of its HED tags expressed in short form. Returned data: a converted JSON sidecar as text or an error file as text if errors.	
sidecar_validate [schema_string, schema_url, schema_version], check_for_warnings	json_string, Validate a BIDS-style JSON sidecar. Returned data: an error file as text if errors.	
spreadsheet_to_long	spreadsheet_string,	

The following table gives an explanation of the parameters used for various services.

Table 5: Parameters for web services.

Key value	Type	Description
check_for_warnings	boolean	If true, check for warnings when processing.
column_x_check:	boolean	If present with value 'on', column x has HED tags.".
column_x_input:	string	Contains the prefix prepended to column x if column x has HED tags.
expand_defs	boolean	If true replaces *def/XXX* with *def-expand/XXX* grouped with the definition content.
events_string	string	A BIDS events file as a string..
hed_columns	list of numbers	A list of HED string column numbers (starting with 1).
hed_schema_string	string	HED schema in XML format as a string.
hed_strings	list of strings	A list containing HED strings.
json_string	string	BIDS-style JSON events sidecar as a string.
json_strings	string	A list of BIDS-style JSON sidecars as strings.
schema_string	string	A HED schema file as a string.
schema_version	string	Version of HED to be accessed if relevant.
service	string	The name of the requested service.
spreadsheet_string	string	A spreadsheet tsv as a string.

6.12.2.3 Service responses

The web-services always return a JSON dictionary with four keys: `service`, `results`, `error_type`, and `error_msg`. If `error_type` and `error_msg` are not empty, the operation failed, while if these fields are empty, the operation completed. Completed operations always return their results in the `results` dictionary. Keys in the `results` dictionary return as part of a HED web service response.

Table 6: The results dictionary.

Key	Type	Description
command	string	Command executed in response to the service request.
command_target	string	The type of data on which the command was executed..
data	string	A list of errors or the processed result.
schema_version	string	The version of the HED schema used in the processing.
msg_category	string	One of success, warning, or failure depending on the result.
msg	string	Explanation of the result of service processing.

The `msg` and `msg_category` pertain to contents of the response information. For example a `msg_category` of `warning` in response to a validation request indicates that the validation completed and that the object that was validated had validation errors. In contrast, the `error_type`, and `error_msg` values are only for web service requests. These keys indicate whether validation was able to take place. Examples of failures that would cause errors include the service timing out or the service request parameters were incorrect.

6.13 CTagger GUI tagging tool

This tutorial is under development.

This tutorial takes you through the process of using CTagger, a user interface we developed to ease the process of adding HED annotations. CTagger can be used as a standalone application or as part of the EEGLAB BIDS data pipeline, making it easy to integrate the annotation pipeline mentioned in the Quick guide.

6.13.1 CTAGGER installation

6.13.1.1 CTAGGER standalone installation

Step 1:Check to see that you have Java installed.

Linux usually comes with OpenJDK (open source version of JDK) already installed. We have tested up to Java version 11 in Mac and Ubuntu.

Executing `java -version` on terminal should return something similar to: `java version "1.8.0_211"` or `openjdk version "11.0.11" 2021-04-20`.

If Java is not installed, download and install Java Runtime Environment accordingly to your OS: https://www.oracle.com/java/technologies/javase-jre8-downloads.html. You might be asked to create an Oracle account first before you can download.

Step 2: Download CTagger.jar.

Step 3: Double-click on *CTagger.jar* to run.

If you're on macOS you might need to update your Security settings to allow the app to run.

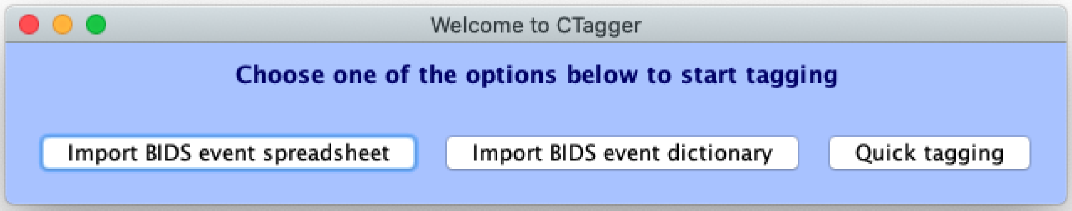
On Linux, you might need to make the jar executable first by executing `chmod +x CTagger.jar` while in the directory containing the downloaded CTagger.

6.13.1.2 CTAGGER in EEGLAB

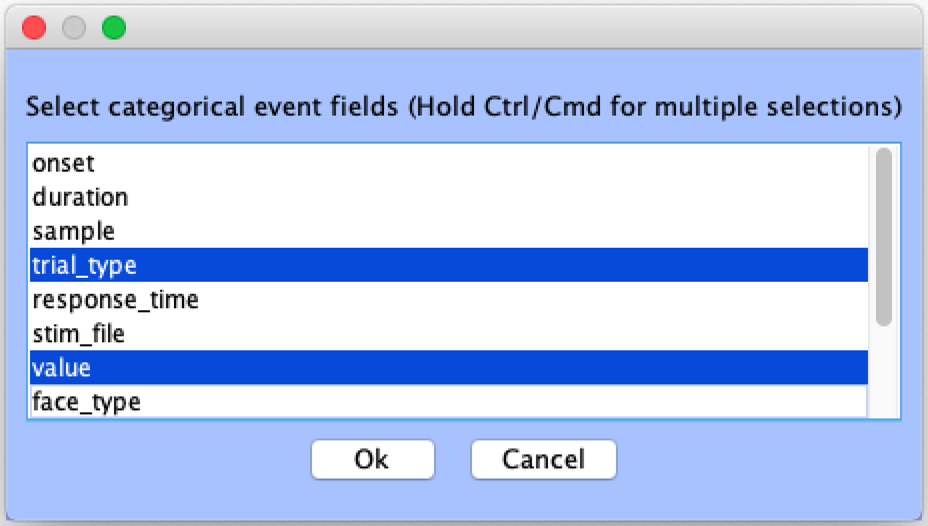
Install HEDTools plugin. Installation is done through the **Manage EEGLAB extensions** options on the EEGLAB GUI File menu. The HEDTools options then appear under the EEGLAB Edit menu once you have loaded a dataset.

6.13.2 Loading BIDS event files

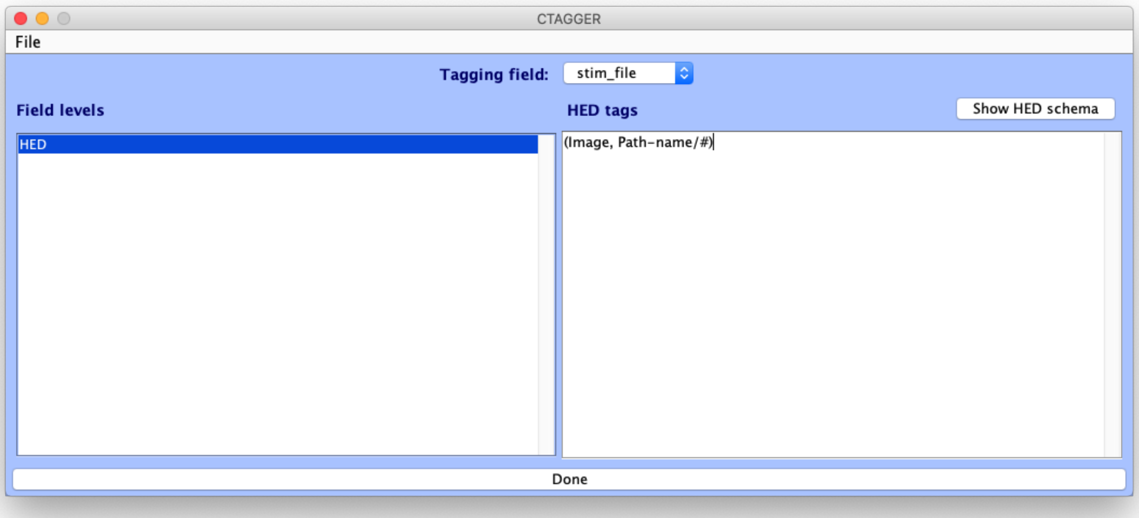
From the CTagger launcher window, users have the option to import either the BIDS event dictionary or event spreadsheet to start tagging.


Welcome to CTagger
Choose one of the options below to start tagging
Import BIDS event spreadsheet
Import BIDS event dictionary
Quick tagging

If **Import BIDS event dictionary** is selected, users will be prompted to select an *events.json* file. Any field (corresponding to event column) with the key "Levels" will be interpreted as having categorical column, and each of the sub levels will be considered as the categorical values of the field. All other fields will be considered as having continuous values. If users import an *events.tsv* file via **Import BIDS event spreadsheet**, CTagger will ask users to specify categorical fields. Unique categorical values of these fields will then be automatically extracted from the file.



Once importing is finished, users will see the main CTagger tagging interface. Users can toggle between different fields using the **Tagging field** dropdown. For non-categorical fields, the left panel will contain a single item **HED** to which the HED string containing the # symbol will be associated as explained above. For categorical fields, the **Field levels** panel on the left will contains a list of the categorical values of the field. Users can choose any item on the list to start tagging. Users specify HED tags on the right panel. CTagger will associated the HED string formed in the right panel with the list item selected in the left panel. It is important that an item is selected from the **Field levels** list, otherwise the HED string formed will not be saved.



(Image, Path-name/#)

You are now ready to start tagging!

6.13.3 Adding HED annotation

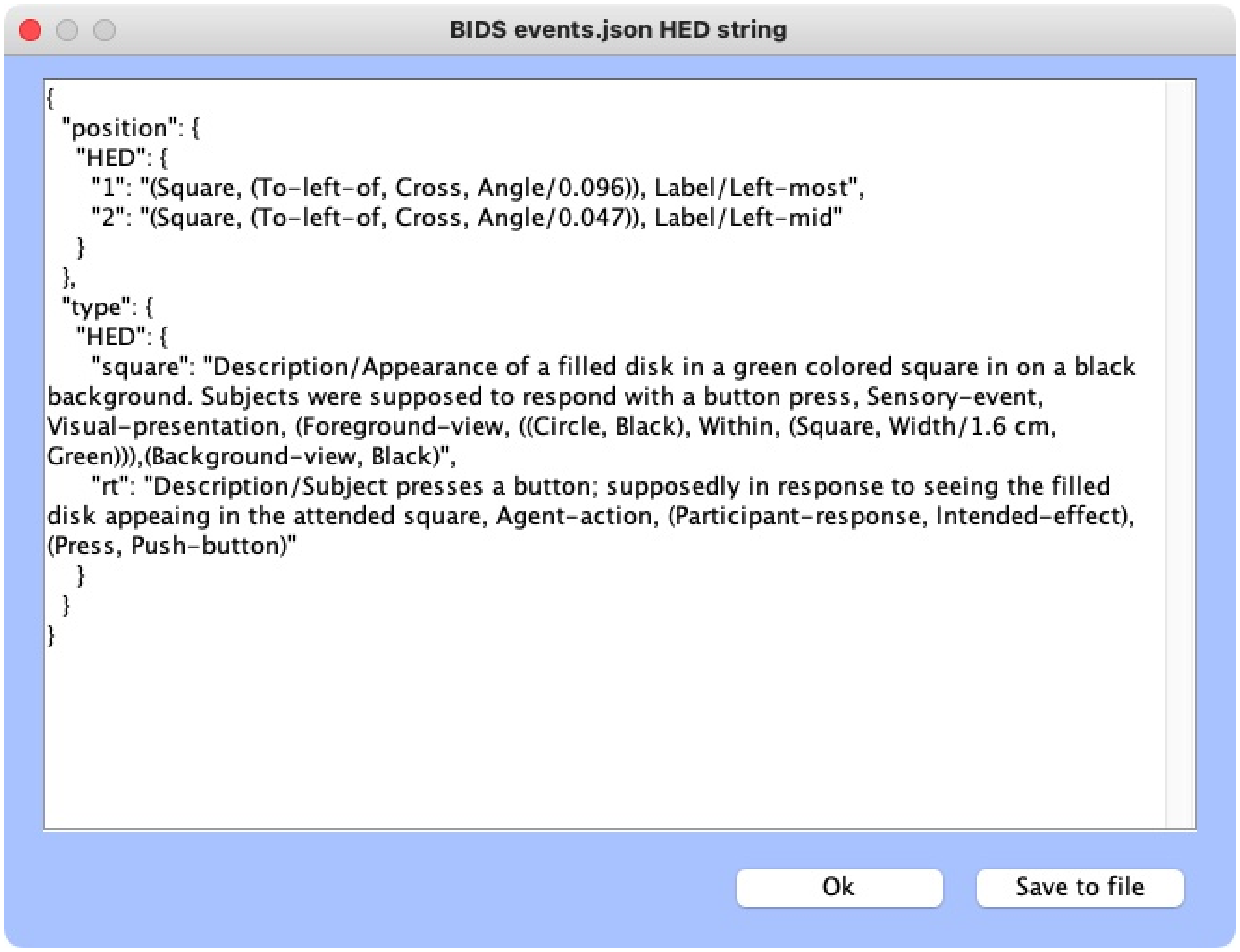
Helping users construct HED annotations quickly and easily is the main goal of CTagger. New **search capacity** and **dynamic formatting** features and the new **schema-browsing view,** help users to quickly compose HED annotations.

When the user types into the tag editor in CTagger, the character sequence is compared to all the schema nodes. Nodes whose long-form tag match any part of the input sequence (case-insensitive) will be displayed (in long form) in the CTagger **Search Results** box (just below the input cursor). Users can scroll through the resulting results display by pressing the *down-arrow* key and clicking on *Enter/return* to select a tag to add to the HED annotation. Else, the user can scroll through the results by mouse and click on a tag to select it. CTagger will always add the short-form tag, replacing the input sequence, and the *Search Results* box will disappear. At any point, the user can make the *Search Results* box disappear by pressing *Esc* key or by clicking on the text area outside the *Search Results* box.

The user can also open up the HED schema itself to browse for an appropriate tag in a more exploratory way, expanding the hierarchical structure of the schema at any point of interest. Clicking on the **Show HED schema** button brings up the schema display, in which any schema node that contains children is expandable/collapsible. Clicking on a node will add its short-form tag to the current annotation. For nodes that take values, clicking on the **#** underneath the node will add the node name, followed by a forward-slash (**/**) to the annotation. the user will then type in the value for that node.

In HED, tags can be grouped together to indicate that they modify each other and should be interpreted (e.g. during search) as a whole unit. A **tag group** is specified by surrounding comma-separated tags in the same group with parentheses. For example, the tag group *(Triangle, Green)* describes a green triangle. HED also allows **nested tag groups**, for example to annotate something containing something else. Having multiple levels of nested tag groups (hence lots of nested pairs of parentheses) can be difficult to parse visually. CTagger allows users to incorporate new lines and tab indentation (using the *Newline* and *Tab* keys) to make their annotations more readable. Once the annotation for an event or event type is complete, a final, comma-separated full-form **HED string** is created. CTagger will automatically strip *Newline* and *Tab* characters from the HED string to produce a HED string format compatible with any annotation destination file.

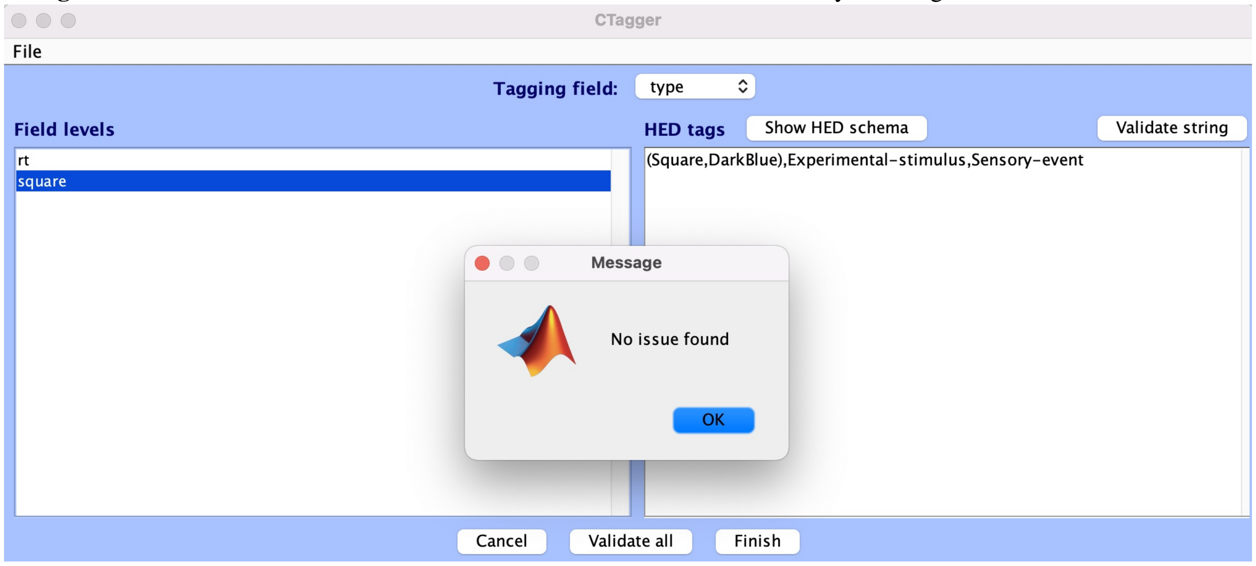
In CTagger, at any time during the annotation process, the user can view the current (if still incomplete) version of the long-form HED string by going to *File > Review all tags*.



```
{
  "position": {
    "HED": {
      "1": "(Square, (To-left-of, Cross, Angle/0.096)), Label/Left-most",
      "2": "(Square, (To-left-of, Cross, Angle/0.047)), Label/Left-mid"
    }
  },
  "type": {
    "HED": {
      "square": "Description/Appearance of a filled disk in a green colored square in on a black
background. Subjects were supposed to respond with a button press, Sensory-event,
Visual-presentation, (Foreground-view, ((Circle, Black), Within, (Square, Width/1.6 cm,
Green))),(Background-view, Black)",
      "rt": "Description/Subject presses a button; supposedly in response to seeing the filled
disk appeing in the attended square, Agent-action, (Participant-response, Intended-effect),
(Press, Push-button)"
    }
  }
}
```

6.13.3.1 Validating your annotation

At any point of the tagging process users can validate their current HED string by clicking on the **Validate string** button. Users can also validate annotation of the entire event file by clicking on the **Validate all** button.



(Square,DarkBlue),Experimental-stimulus,Sensory-event

No issue found

Users must fix any validation error before being able to finish tagging. Once done with their annotation, users can then copy the HED string and paste it into an intended location (e.g., an **event design** file table), or can save the HED annotation as a JSON dictionary file. Click **Finish > Save to file** and give a name to your json file; then select where you want to save it. You're done!

6.14 File remodeling tools

Remodeling refers to the process of transforming a tabular file into a different form in order to disambiguate the information or to facilitate a particular analysis. The remodeling operations are specified in a JSON (`.json`) file, giving a record of the transformations performed.

There are two types of remodeling operations: **transformation** and **summarization**. The **transformation** operations modify the tabular files, while **summarization** produces an auxiliary information file but leaves the tabular files unchanged.

The file remodeling tools can be applied to any tab-separated value (`.tsv`) file but are particularly useful for restructuring files representing experimental events. Please read the ***File remodeling quickstart*** tutorials for an introduction and basic use of the tools.

The file remodeling tools can be applied to individual files using the **HED online tools** or to entire datasets using the ***remodel command-line interface*** either by calling Python scripts directly from the command line or by embedding calls in a Jupyter notebook. The tools are also available as ***HED RESTful services***. The online tools are particularly useful for debugging.

This user's guide contains the following topics:

- ***Overview of remodeling***
- ***Installing the remodel tools***
- ***Remodel command-line interface***
- ***Remodel scripts***
  - ***Backing up files***
  - ***Remodeling files***
  - ***Restoring files***
- ***Remodel with HED***
- ***Remodel sample files***
  - ***Sample remodel file***
  - ***Sample remodel event file***
  - ***Sample remodel sidecar file***
- ***Remodel transformations***
  - ***Factor column***
  - ***Factor HED tags***
  - ***Factor HED type***
  - ***Merge consecutive***
  - ***Remap columns***
  - ***Remove columns***

  - *Remove rows*
  - *Rename columns*
  - *Reorder columns*
  - *Split rows*
- *Remodel summarizations*
  - *Summarize column names*
  - *Summarize column values*
  - *Summarize definitions*
  - *Summarize sidecar from events*
  - *Summarize hed tags*
  - *Summarize hed type*
  - *Summarize hed validation*
- *Remodel implementation*

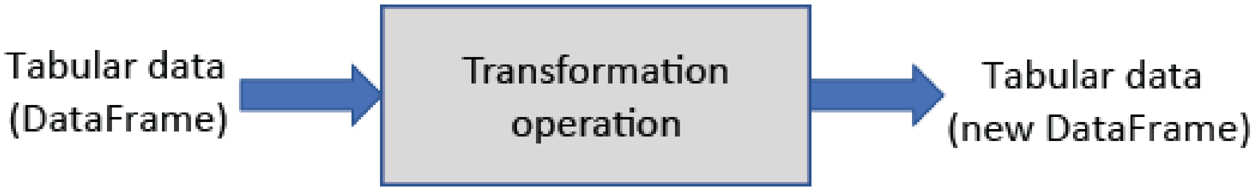
6.14.1 Overview of remodeling

Remodeling consists of restructuring and/or extracting information from tab-separated value files based on a specified list of operations contained in a JSON file.

Internally, the remodeling operations represent the tabular file using a **Pandas DataFrame**.

6.14.1.1 Transformation operations

Transformation operations, shown schematically in the following figure, are designed to transform an incoming tabular file into a new DataFrame without modifying the incoming data.



Transformation operations are stateless and do not save any context information or affect future applications of the transformation.

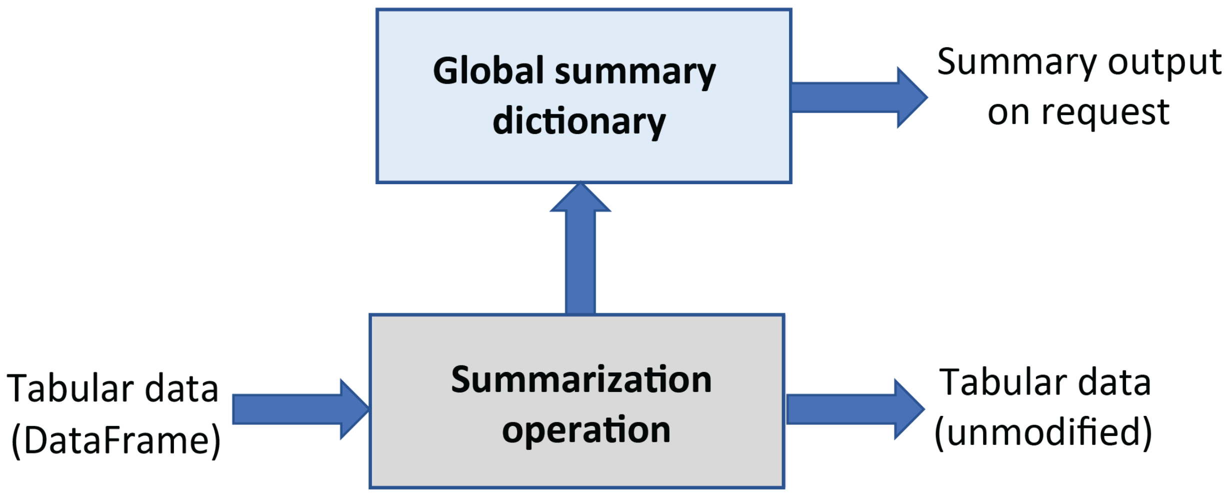
Transformations, themselves, do not have any output and just return a new, transformed DataFrame. In other words, transformations do not operate in place on the incoming DataFrame, but rather, they create a new DataFrame containing the result.

Typically, the calling program is responsible for reading and saving the tabular file, so the user can choose whether to overwrite or create a new file.

See the ***remodeling tool program interface*** section for information on how to call the operations.

6.14.1.2 Summarization operations

Summarization operations do not modify the input DataFrame but rather extract and save information in an internally stored summary dictionary as shown schematically in the following figure.


Global summary dictionary
Summary output on request
Tabular data (DataFrame)
Summarization operation
Tabular data (unmodified)

The dispatcher that executes remodeling operations can be interrogated at any time for the state information contained in the global summary dictionary and can save additional summary information at any time during execution. Usually summaries are dumped at the end of processing to the `derivatives/remodel/summaries` subdirectory under the dataset root.

Summarization operations may appear anywhere in the operation list, and the same type of summary may appear multiple times under different names in order to track progress.

The dispatcher stores information from each uniquely named summarization operation as a separate summary dictionary entry. Within its summary information, most summarization operations keep a separate summary for each individual file and have methods to create an overall summary of the information for all the files that have been processed by the summarization.

Summarization results are available in JSON (`.json`) and text (`.txt`) formats.

6.14.1.3 Available operations

The following table lists the available remodeling operations with brief example use cases and links to further documentation. Operations not listed in the summarize section are transformations.

Table 7: Summary of the HED remodeling operations for tabular files.

Category	Operation	Example use case
clean-up		
	remove_columns	Remove temporary columns created during restructuring.
	remove_rows	Remove rows with n/a values in a specified column.
	rename_columns	Make columns names consistent across a dataset.
	reorder_columns	Make column order consistent across a dataset.
factor		
	factor_column	Extract factor vectors from a column of condition variables.
	factor_hed_tags	Extract factor vectors from search queries of HED annotations.
	factor_hed_type	Extract design matrices and/or condition variables.
restructure		
	merge_consecutive	Replace multiple consecutive events of the same typewith one event of longer duration.
	remap_columns	Create m columns from values in n columns (for recoding).
	split_rows	Split trial-encoded rows into multiple events.
summarize		
	summarize_column_names	Summarize column names and order in the files.
	summarize_column_values	Count the occurrences of the unique column values.
	summarize_sidecar_from_events	Generate a sidecar template from an event file.
	summarize_hed_tags	Summarize the HED tags present in the HED annotations for the dataset.
	summarize_hed_type	Summarize the detailed usage of a particular type tag such as *Condition-variable* or *Task* (used to automatically extract experimental designs).
	summarize_hed_validation	Validate the data files and report any errors.

The **clean-up** operations are used at various phases of restructuring to assure consistency across dataset files.

The **factor** operations produce column vectors with the same number of rows as the data file from which they were calculated. They encode condition variables, design matrices, or other search criteria. See the ***HED conditions and design matrices*** for more information on factoring and analysis.

The **restructure** operations modify the way in which a data file represents its information.

The **summarize** operations produce dataset-wide summaries of various aspects of the data files as well as summaries of the individual files.

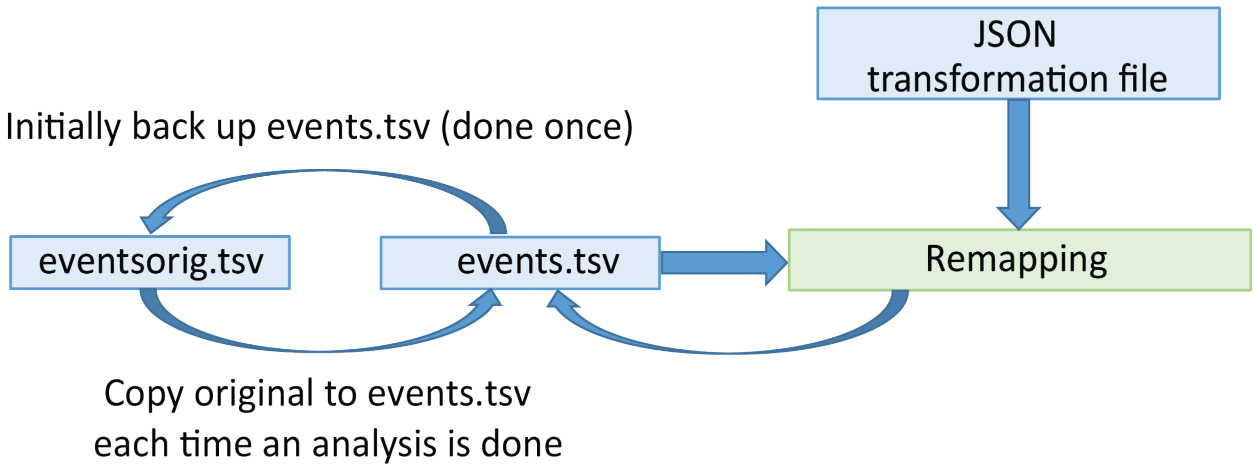
6.14.2 Installing the remodel tools

The remodeling tools are available in the GitHub **hed-python** repository along with other tools for data cleaning and curation. Although version 0.1.0 of this repository is available on **PyPI** as `hedtools`, the version containing the restructuring tools (Version 0.2.0) is still under development and has not been officially released. However, the code is publicly available on the `develop` branch of the hed-python repository and can be directly installed from GitHub using `pip`:

```
pip install git+https://github.com/hed-standard/hed-python/@develop
```

The web services and online tools supporting remodeling are available on the **HED online tools dev server**. When version 0.2.0 of `hedtools` is officially released on PyPI, restructuring will become available on the released **HED online tools**. A docker version is also under development.

The following diagram shows a schematic of the remodeling process.


JSON transformation file
Initially back up events.tsv (done once)
eventsorig.tsv
events.tsv
Remapping
Copy original to events.tsv each time an analysis is done

Initially, the user creates a backup of the specified tabular files (usually `events.tsv` files). This backup is a mirror of the data files in the dataset, but is located in the `derivatives/remodel/backups` directory and never modified once the backup is created.

Remodeling applies a sequence of operations specified in a JSON remodel file to the backup versions of the data files. The JSON remodel file provides a record of the operations performed on the file. If the user detects a mistake in the transformations, he/she can correct the transformation file and rerun the transformations.

Remodeling always runs on the original backup version of the file rather than the transformed version, so the transformations can always be corrected and rerun. It is possible to by-pass the backup, particularly if only using summarization operations, but this is not recommended and should be done with care.

6.14.3 Remodel command-line interface

The remodeling toolbox provides Python scripts with command-line interfaces to create or restore backups and to apply operations to the files in a dataset. The file remodeling tools may be applied to datasets that are in free form under a directory root or that are in **BIDS-format**. Some operations use ***HED (Hierarchical Event Descriptors)*** annotations. See the ***Remodel with HED*** section for a discussion of these operations and how to use them.

The remodeling command-line interface can be used from the command line, called from another Python program, or used in a Jupyter notebooks. Example notebooks can be found in the **Jupyter notebooks** to support remodeling.

6.14.3.1 Calling remodel tools

The remodeling tools provide three Python programs for backup (`run_remodel_backup`), remodeling (`run_remodel`) and restoring (`run_remodel_restore`) event files. These programs can be called from the command line or from another Python program.

The programs use a standard command-line argument list for specifying input as summarized in the following table.

Table 8: Summary of command-line arguments for the remodeling programs.

Script name	Arguments	Purpose
run_remodel_backup	*data_dir-e --extensions-f --file-suffix-n --backup-name-t --task-names-v --verbose-w --work-dir-x --exclude-dirs*	Create a backup event files.
run_remodel	*data_dirmodel_path-b --bids-format-e --extensions-f --file-suffix-i --individual-summaries-j --json-sidecar-n --backup-name-nb --no-backup-ns --no-summaries-nu --no-update-r --hed-version-s --save-formats-t --task-names-v --verbose-w --work-dir-x --exclude-dirs*	Restructure or summarize the event files.
run_remodel_restore	*data_dir-n --backup-name-t --task-names-v --verbose-w --work-dir*	Restore a backup of event files.

All the scripts have a required argument, which is the full path of the dataset root (*data_dir*). The `run_remodel` program has a required parameter which is the full path of a JSON file containing a specification of the remodeling commands to be run.

6.14.3.2 Remodel command-line arguments

This section describes the arguments that are used for the remodeling command-line interface with examples and more details.

Positional arguments

Positional arguments are required and must be given in the order specified.

`data_dir`

The full path of dataset root directory.

`model_path`

The full path of the JSON remodel file (for *run_remodel* only).

Named arguments

Named arguments consist of a key starting with a hyphen and are possibly followed by a value. Named arguments can be given in any order or omitted. If omitted, a specified default is used. Argument keys and values are separated by spaces.

For argument values that are lists, the key is given followed by the items in the list, all separated by spaces.

Each command has two different forms of the key name: a short form (a single hyphen followed by a single character) and a longer form (two hyphens followed by a more self-explanatory name). Users are free to use either form.

`-b`, `--bids-format`

If this flag present, the dataset is in BIDS format with sidecars. Tabular files are located using BIDS naming.

`-e`, `--extensions`

This option is followed by a list of file extension(s) of the data files to process. The default is `.tsv`. Comma separated tabular files are not permitted.

`-f`, `--file-suffix`

This option is followed by the suffix names of the files to be processed. For example `events` (the default) captures files named `events.tsv` if the default extension is used. The filename without the extension must end in one of the specified suffixes in order to be backed up or transformed.

`-i`, `--individual-summaries`

This option offers a choice among three options:

- `separate`: Individual summaries for each file in separate files in addition the overall summary.
- `consolidated`: Individual summaries written in the same file as the overall summary.
- `none`: Only an overall summary.

`-j`, `--json-sidecar`

This option is followed by the full path of the JSON sidecar with HED annotations to be applied during the processing of HED-related remodeling operations.

`-n`, `--backup-name`

The name of the backup used for the remodeling (default: `default_back`).

`-nb`, `--no-backup`

If present, no backup is used. Rather operations are performed directly on the files.

`-ns`, `--no-summaries`

If present, no summary files are output.

`-nu`, `--no-update`

If present, the modified files are not output.

`-r`, `--hed-versions`

This option is followed by one or more HED versions. Versions of the standard schema are specified by their semantic versions (e.g., `8.1.0`), while library schema versions are prefixed by their library name (e.g., `score_1.0.0`).

If more than one HED schema version is given, all but one of the versions must start with an additional namespace designator (e.g., `sc:`). At most one version can omit the namespace designator when multiple schema are being used. In annotations, tags must start with the namespace designator of the corresponding schema from which they were selected (e.g. `sc:Sleep-modulator` if the SCORE library was designated by `sc:score_1.0.0`).

`-s`, `--save-formats`

This option is followed by the extensions (including .) of the formats in which to save summaries (default: `.txt .json`).

`-t`, `--task-names`

The name(s) of the tasks to be included (for BIDS-formatted files only). When a dataset includes multiple tasks, the event files are often structured differently for each task and thus require different transformation files. This option allows the backups and operations to be restricted to a single task. If this option is omitted, all tasks are used.

`-v`, `--verbose`

If present, more comprehensive messages documenting transformation progress are printed to standard output.

`-w`, `--work-dir`

The path to the remodeling work root directory –both for backups and summaries (default: `[data_root]/derivatives/remodel`). Use the `-nb` option if you wish to omit the backup (in `run_remodel`).

`-x`, `--exclude-dirs`

The directories to exclude when gathering the data files to process. For BIDS datasets, these are typically `derivatives`, `stimuli`, and `sourcecode`. Any subdirectory with a path component named `remodel` is automatically excluded from remodeling, as these directories are reserved for storing backup, state, and result information for the remodeling process itself.

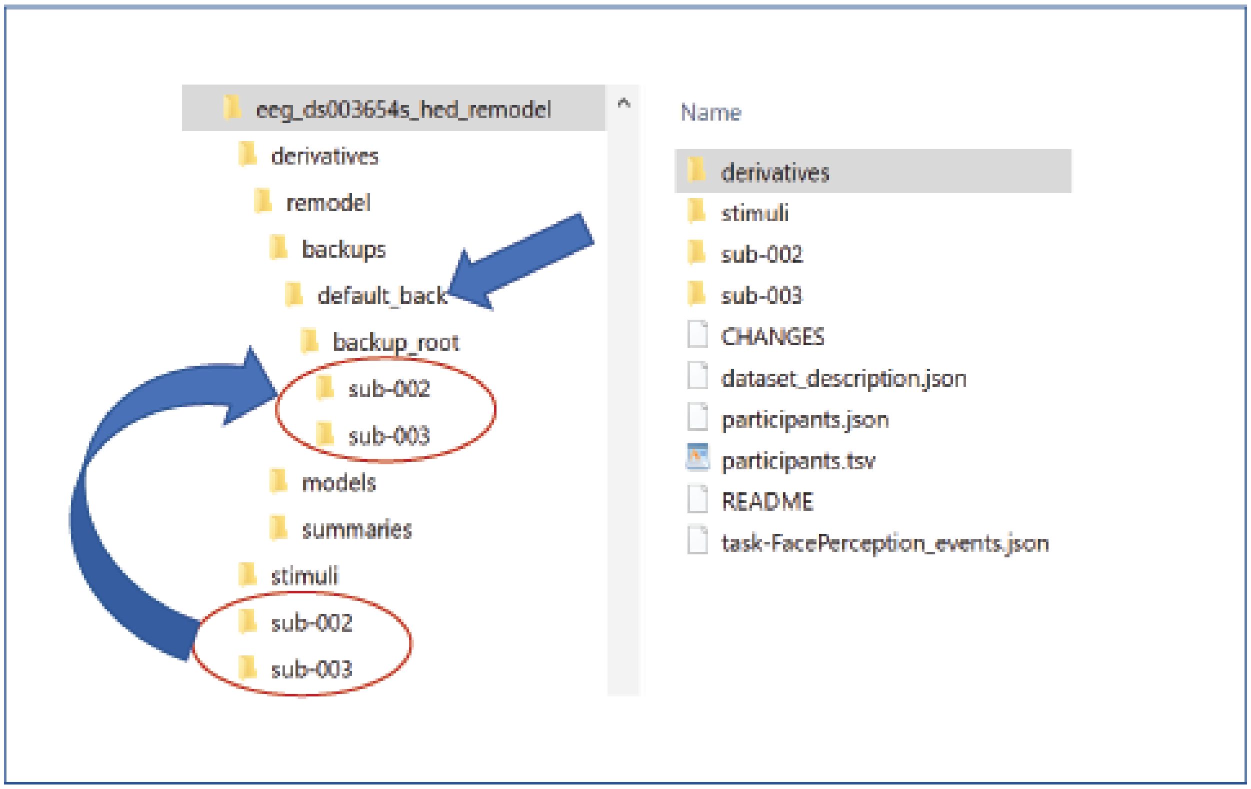
6.14.4 Remodel scripts

This section discusses the three main remodeling scripts with command-line interfaces to support backup, remodeling, and restoring the tabular files used in the remodeling process. These scripts can be run from the command line or from another Python program using a function call.

6.14.4.1 Backing up files

The `run_remodel_backup` Python program creates a backup of the specified files. The backup is always created in the `derivatives/remodel/backups` subdirectory under the dataset root as shown in the following example for the sample dataset `eeg_ds003645s_hed_remodel`, which can be found in the `datasets` subdirectory of the **hed-examples** GitHub repository.

Directory structure of remodeling backup



The backup process creates a mirror of the directory structure of the source files to be backed up in the directory `derivatives/remodel/backups/backup_name/backup_root` as shown in the figure above. The default backup name is `default_back`.

In the above example, the backup has subdirectories `sub-002` and `sub-003` just like the main directory of the dataset. These subdirectories only contain backups of the files to be transformed (by default files with names ending in `events.tsv`).

In addition to the `backup_root`, the backup directory also contains a dictionary of backup files in the `backup_lock.json` file. This dictionary is used internally by the remodeling tools. The backup should be created once and not modified by the user.

The following example shows how to run the `run_remodel_backup` program from the command line to back up the dataset located at `/datasets/eeg_ds003645s_hed_remodel`.

Example of calling run_remodel_backup from the command line.

```
python run_remodel_backup /datasets/eeg_ds003645s_hed_remodel -x derivatives stimuli
```

Since the `-f` and `-e` arguments are not given, the default file suffix and extension values apply, so only files of the form `events.tsv` are backed up. The `-x` option excludes any source files from the `derivatives` and `stimuli` subdirectories. These choices can be overridden using additional command-line arguments.

The following shows how the `run_remodel_backup` program can be called from a Python program or a Jupyter notebook. The command-line arguments are given in a list instead of on the command line.

Example of Python code to call run_remodel_backup using a function call.

```
import hed.tools.remodeling.cli.run_remodel_backup as cli_backup

data_root = '/datasets/eeg_ds003645s_hed_remodel'
arg_list = [data_root, '-x', 'derivatives', 'stimuli']
cli_backup.main(arg_list)
```

During remodeling, each file in the source is associated with a backup file using its relative path from the dataset root. Remodeling is performed by reading the backup file, performing the operations specified in the JSON remodel file, and overwriting the source file as needed.

Users can also create alternatively named backups by providing the `-n` argument with a backup name to the `run_remodel_backup` program. To use backup files from another named backup, call the remodeling program with the `-n` argument and the correct backup name. Named backups can provide checkpoints to allow the execution of transformations to start from intermediate points.

NOTE: You should not delete backups, even if you have created multiple named backups. The backups provide useful state and provenance information about the data.

6.14.4.2 Remodeling files

Remodeling consists of applying a sequence of operations from the ***remodel operation summary*** to successively transform each backup file according to the instructions and to overwrite the actual files with the final result.

If the dataset has no backups, the actual data files rather than the backups are transformed. You are expected to ***create the backup*** (just once) before running the remodeling operations. Going without backup is not recommended unless you are only doing summarization operations.

The operations are specified as a list of dictionaries in a JSON file in the ***remodel sample files*** as discussed below.

Before running remodeling transformations on an entire dataset, consider using the **HED online tools** to debug your remodeling operation file on a single file. The remodeling process always starts with the original backup files, so the usual development path is to incrementally add operations to the end of your transformation JSON file as you develop and test on a single file until you have the desired end result.

The following example shows how to run a remodeling script from the command line. The example assumes that the backup has already been created for the dataset.

Example of calling run_remodel from the command line.

```
python run_remodel /datasets/eeg_ds003645s_hed_remodel /datasets/remove_extra_rmdl.json -
↪x derivatives simuli
```

The script has two required arguments the dataset root and the path to the JSON remodel file. Usually, the JSON remodel files are stored with the dataset itself in the `derivatives/remodel/remodeling_files` subdirectory, but common scripts can be stored in a central place elsewhere.

The additional keyword option, `-x` in the example indicates that directory paths containing the component `derivatives` or the component `stimuli` should be excluded. Excluded directories need not have their excluded path component at the top level of the dataset. Subdirectory paths containing the `remodel` path component are automatically excluded.

The command-line interface can also be used in a Jupyter notebook or as part of a larger Python program by calling the `main` function with the equivalent command-line arguments provided in a list with the positional arguments appearing first.

The following example shows Python code to remodel a dataset using the command-line interface. This code can be used in a Jupyter notebook or in another Python program.

Example Python code to call run_remodel using a function call.

```python
import hed.tools.remodeling.cli.run_remodel as cli_remodel

data_root = '/datasets/eeg_ds003645s_hed_remodel'
model_path = '/datasets/remove_extra_rmdl.json'
arg_list = [data_root, model_path, '-x', 'derivatives', 'stimuli']
cli_remodel.main(arg_list)
```

6.14.4.3 Restoring files

Since remodeling always uses the backed up version of each data file, there is no need to restore these files to their original state between remodeling runs. However, when finished with an analysis, you may want to restore the data files to their original state.

The following example shows how to call `run_remodel_restore` to restore the data files from the default backup. The restore operation restores all the files in the specified backup.

Example of calling run_remodel_restore from the command line.

```
python run_remodel_restore /datasets/eeg_ds003645s_hed_remodel
```

As with the other command-line programs, `run_remodel_restore` can be also called using a function call.

Example Python code to call *run_remodel_restore* using a function call.

```python
import hed.tools.remodeling.cli.run_restore as cli_remodel

data_root = '/datasets/eeg_ds003645s_hed_remodel'
cli_remodel.main([data_root])
```

6.14.5 Remodel with HED

HED (Hierarchical Event Descriptors) is a system for annotating data in a manner that is both human-understandable and machine-actionable. HED provides much more detail about the events and their meanings, If you are new to HED see the *HED annotation quickstart*. For information about HED's integration into BIDS (Brain Imaging Data Structure) see *BIDS annotation quickstart*.

Currently, five remodeling operations rely on HED annotations:

- *factor_hed_tags*
- *factor_hed_type*
- *summarize_hed_tags*
- *summarize_hed_type*
- *summarize_hed_validation*.

HED tags provide a mechanism for advanced data analysis and for extracting experiment-specific information from the data files. However, since HED information is not always stored in the data files themselves, you may need to provide a HED schema and a JSON sidecar.

The HED schema defines the allowed HED tag vocabulary, and the JSON sidecar associates HED annotations with the information in the columns of the event files. If you are not using any of the HED operations in your remodeling, you do not have to provide this information.

6.14.5.1 Extracting HED information from BIDS

The simplest way to use HED with `run_remodel` is to use the `-b` option, which indicates that the dataset is in **BIDS** (Brain Imaging Data Structure) format.

BIDS is a standardized way of organizing neuroimaging data. HED and BIDS are well integrated. If you are new to BIDS, see the *BIDS annotation quickstart*.

A HED-annotated BIDS dataset provides the HED schema version in the `dataset_description.json` file located directly under the BIDS dataset root.

BIDS datasets must have filenames in a specific format, and the HED tools can locate the relevant JSON sidecars for each data file based on this information.

6.14.5.2 Directly specifying HED information

If your data is already in BIDS format, using the `-b` option is ideal since the needed information can be located automatically. However, early in the experimental process, your datafiles are not likely to be organized in BIDS format, and this option will not be available if you want to use HED.

Without the `-b` option, the remodeling tools locate the appropriate files based on specified filename suffixes and extensions. In order to use HED operations, you must explicitly specify the HED versions using the `-r` option. The `-r` option supports a list of HED versions if multiple HED schemas are used. For example: `-r 8.1.0 sc:score_1.0.0` specifies that vocabulary will be drawn from standard HED Version 8.1.0 and from HED SCORE library version 1.0.0. Annotations containing tags from SCORE should be prefixed with `sc:`. Note: both of the schemas can be viewed by the **HED Schema Browser**.

Usually, annotators will consolidate HED annotations in a single JSON sidecar file located at the top-level of the dataset. The path of this sidecar can be passed as a command-line argument using the `-j` option. If more than one JSON sidecar file contains HED annotations, users will need to call the lower-level remodeling functions to perform these operations.

The following example illustrates a command-line call that passes both a HED schema version and the path to the JSON file with the HED annotations.

Remodeling a non-BIDS dataset using HED.

```
python run_remodel /datasets/eeg_ds003645s_hed_remodel /datasets/summarize_conditions_
↪rmdl.json \
-x derivatives simuli -r 8.1.0 -j /datasets/eeg_ds003645s_hed_remodel/task-
↪FacePerception_events.json
```

Example Python code to use run_remodel on a non-BIDS dataset.

```
import hed.tools.remodeling.cli.run_remodel as cli_remodel

data_root = '/datasets/eeg_ds003645s_hed_remodel'
model_path = '/datasets/summarize_conditions_rmdl.json'
json_path = '/datasets/eeg_ds003645s_hed_remodel/task-FacePerception_events.json'
arg_list = [data_root, model_path, '-x', 'derivatives', 'stimuli', '-r' 8.1.0 '-j' json_
↪path]
cli_remodel.main(arg_list)
```

6.14.6 Remodel error handling

Errors can occur during several stages in during remodeling and how they are handled depends on the type of error and where the error occurs. Except for the validation summary, the underlying remodeling code raises exceptions for most errors.

6.14.6.1 Errors in the remodel file

Each individual operation raises an exception if required parameters are missing or the values provided for the parameters are of the wrong type. However, the higher-level calling mechanisms provided through `run_remodel` call the `parse_operations` static method provided by the `Dispatcher` to create a parsed operation list. This call either returns a list of parsed operations or a list of parse errors for the operations in the list.

If there are any errors in the remodel file, no operations are run, but the errors for all operations in the list are reported. This allows users to correct errors in all operations in one pass without any data modification. The **HED online tools** are particularly useful for debugging the syntax and other issues in the remodeling process.

6.14.6.2 Execution-time remodel errors

When an error occurs during execution, an exception is raised. Exceptions are raised for invalid or missing files or if a transformed file is unable to be rewritten due to improper file permissions. Each individual operation may also raise an exception if the data file being processed does not have the expected information, such as a column with a particular name.

Exceptions raised during execution cause the process to be terminated and no further files are processed.

6.14.7 Remodel sample files

All remodeling operations are specified in a standardized JSON remodel input file. The following shows the contents of the JSON remodeling file `remove_extra_rmdl.json`, which contains a single operation with instructions to remove the `value` and `sample` columns from the data file if the columns exist.

6.14.7.1 Sample remodel file

A sample JSON remodeling file with a single remove_columns transformation operation.

```
[
    {
        "operation": "remove_columns",
        "description": "Remove unwanted columns prior to analysis",
        "parameters": {
            "remove_names": ["value", "sample"],
            "ignore_missing": true
        }
    }
]
```

Each operation is specified in a dictionary with three top-level keys: "operation", "description", and "parameters". The value of the "operation" is the name of the operation. The "description" value should include the reason this operation was needed, not just a description of the operation itself. Finally, the "parameters" value is a dictionary mapping parameter name to parameter value.

The parameters for each operation are listed in ***Remodel transformations*** and ***Remodel summarizations*** sections. An operation may have both required and optional parameters. Optional parameters may be omitted if unneeded, but all parameters are specified in the "parameters" section of the dictionary.

The remodeling JSON files should have names ending in `_rmdl.json` to more easily distinguish them from other JSON files. Although these files can be stored anywhere, their preferred location is in the `deriviatves/remodel/models` subdirectory under the dataset root so that they can provide provenance for the dataset.

6.14.7.2 Sample remodel event file

Several examples illustrating the remodeling operations use the following excerpt of the stop-go task from sub-0013 of the AOMIC-PIOP2 dataset available on **OpenNeuro** as ds002790. The full event file is **sub-0013_task-stopsignal_acq-seq_events.tsv**.

Excerpt from an event file from the stop-go task of AOMIC-PIOP2 (ds002790).

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex
0.0776	0.5083	go	n/a	0.565		correct	right
5.5774	0.5083	unsuccesful_stop	0.2	0.49	correct	right	female
9.5856	0.5084	go	n/a	0.45	correct	right	female
13.5939	0.5083	succesful_stop	0.2	n/a	n/a	n/a	female
17.1021	0.5083	unsuccesful_stop	0.25	0.633	correct	left	male
21.6103	0.5083	go	n/a	0.443	correct	left	male

6.14.7.3 Sample remodel sidecar file

For remodeling operations that use HED, a JSON sidecar is usually required to provide the necessary HED annotations. The following JSON sidecar excerpt is used in several examples to illustrate some of these operations. The full JSON file can be found at **task-stopsiqnal_acq-seq_events.json**.

Excerpt of JSON sidecar with HED annotations for the stop-go task of AOMIC-PIOP2.

```
{
    "trial_type": {
        "HED": {
            "succesful_stop": "Sensory-presentation, Visual-presentation, Correct-action,
↪ Image, Label/succesful_stop",
            "unsuccesful_stop": "Sensory-presentation, Visual-presentation, Incorrect-
↪action, Image, Label/unsuccesful_stop",
            "go": "Sensory-presentation, Visual-presentation, Image, Label/go"
        }
    },
    "stop_signal_delay": {
        "HED": "(Auditory-presentation, Delay/# s)"
        },
    "sex": {
        "HED": {
            "male": "Def/Male-image-cond",
            "female": "Def/Female-image-cond"
        }
    },
    "hed_defs": {
        "HED": {
            "def_male": "(Definition/Male-image-cond, (Condition-variable/Image-sex,␣
↪(Male, (Image, Face))))",
            "def_female": "(Definition/Female-image-cond, (Condition-variable/Image-sex,␣
↪(Female, (Image, Face))))"
        }
    }
}
```

Notice that the JSON file has some keys (e.g., "trial_type", "stop_signal_delay", and "sex") which also correspond to columns in the events file. The "hed_defs" key corresponds to an extra entry in the JSON file that, in this case, provides the definitions needed in the HED annotation.

HED operations also require the HED schema. Most of the examples use HED standard schema version 8.1.0.

6.14.8 Remodel transformations

6.14.8.1 Factor column

The *factor_column* operation appends factor vectors to tabular files based on the values in a specified file column. Each factor vector contains a 1 if the corresponding row had that column value and a 0 otherwise. The *factor_column* is used to reformat event files for analyses such as linear regression based on column values.

Factor column parameters

Parameters for the *factor_column* operation.

Parameter	Type	Description
column_name	str	The name of the column to be factored.
factor_values	list	Column values to be included as factors.
factor_names	list	Column names for created factors.

If *column_name* is not a column in the data file, a `ValueError` is raised.

If *factor_values* is empty, factors are created for each unique value in *column_name*. Otherwise, only factors for the specified column values are generated. If a specified value is missing in a particular file, the corresponding factor column contains all zeros.

If *factor_names* is empty, the newly created columns are of the form *column_name.factor_value*. Otherwise, the newly created columns have names *factor_names*. If *factor_names* is not empty, then *factor_values* must also be specified and both lists must be of the same length.

Factor column example

The *factor_column* operation in the following example specifies that factor columns should be created for *succesful_stop* and *unsuccesful_stop* of the *trial_type* column. The resulting columns are called *stopped* and *stop_failed*, respectively.

A sample JSON file with a single *factor_column* transformation operation.

```
[{
    "operation": "factor_column"
    "description": "Create factors for the succesful_stop and unsuccesful_stop values."
    "parameters": {
        "column_name": "trial_type",
        "factor_values": ["succesful_stop", "unsuccesful_stop"],
        "factor_names": ["stopped", "stop_failed"]
```

(continues on next page)

(continued from previous page)

```
    }
}]
```

The results of executing this *factor_column* operation on the ***sample remodel event file*** are:

Results of the factor_column operation on the sampe data.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex	stopped	stop_failed
0.0776	0.5083	go	n/a	0.565	correct	right	female	0	0
5.5774	0.5083	unsuccessful_stop	0.2	0.49	correct	right	female	0	1
9.5856	0.5084	go	n/a	0.45	correct	right	female	0	0
13.5939	0.5083	successful_stop	0.2	n/a	n/a	n/a	female	1	0
17.1021	0.5083	unsuccessful_stop	0.25	0.633	correct	left	male	0	1
21.6103	0.5083	go	n/a	0.443	correct	left	male	0	0

6.14.8.2 Factor HED tags

The *factor_hed_tags* operation is similar to the *factor_column* operation in that it produces factor vectors containing 0's and 1, which are appended to the returned DataFrame. However, rather than basing these vectors on values in a specified column, the factors are computed by determining whether the assembled HED annotations for each row satisfies a specified search query.

An example search query is whether the assembled HED annotation contains a particular HED tag. The ***HED search guide*** tutorial discusses the HED search facility in more detail.

Factor HED tags parameters

Parameters for the *factor_hed_tags* operation.

Parameter	Type	Description
queries	list	A list of HED query strings.
query_names	list	A list of names for the resulting factor columns generated by the queries.
remove_types	list	Structural HED tags to be removed (usually `Condition-variable` and `Task`).
expand_context	bool	(Optional) Expand the context and remove `Onse` and`Offset` tags before the query.

The *query_names* list, which must be empty or the same length as *queries*, contains the names of the factor columns produced by the search. If the *query_names* list is empty, the result columns are titled "query_1", "query_2", etc.

The *remove_types* and *expand_context* are not yet implemented, and hence ignored in the current release.

Factor HED tags example

The *factor_hed-tags* operation in the following example produce two factor columns with 1's where the HED string for a row contains the `Correct-action` and `Incorrect-action`, respectively. The resulting factor columns are named *correct* and *incorrect*, respectively.

A sample JSON file with a single *factor_hed_tags* transformation operation.

```
[{
    "operation": "factor_hed_tags"
    "description": "Create factors based on whether the event represented a correct or␣
↪incorrect action.",
    "parameters": {
        "queries": ["correct-action", "incorrect-action"],
        "query-names": ["correct", "incorrect"],
        "remove-types": [],
        "expand_context": false
    }
}]
```

The results of executing this *factor_hed-tags* operation on the ***sample remodel event file*** using the ***sample remodel sidecar file*** for HED annotations is:

Results of *factor_hed_tags*.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex	correct	incorrect
0.0776	0.5083	go	n/a	0.565	correct	right	female	0	0
5.5774	0.5083	unsuccesful_stop	0.2	0.49	correct	right	female	0	1
9.5856	0.5084	go	n/a	0.45	correct	right	female	0	0
13.5939	0.5083	succesful_stop	0.2	n/a	n/a	n/a	female	1	0
17.1021	0.5083	unsuccesful_stop	0.25	0.633	correct	left	male	0	1
21.6103	0.5083	go	n/a	0.443	correct	left	male	0	0

6.14.8.3 Factor HED type

The *factor_hed_type* operation produces factor columns based on values of the specified HED type tag. The most common type is the HED *Condition-variable* tag, which corresponds to factor vectors based on the experimental design. Other commonly use type tags include *Task*, *Control-variable*, and *Time-block*.

We assume that the dataset has been annotated using HED tags to properly document information such as experimental conditions, and focus on how such an annotated dataset can be used with remodeling to produce factor columns corresponding to these type variables.

For additional information on how to encode experimental designs using HED, see ***HED conditions and design matrices***.

Factor HED type parameters

Parameters for *factor_hed_type* operation.

Parameter	Type	Description
type_tag	str	HED tag used to find the factors (most commonly *Condition-variable*).
type_values	list	Values to factor for the *type_tag*.If empty, all values of that *type_tag* are used.

Factor HED type example

The *factor_hed_type* operation in the following example appends additional columns to each data file corresponding to each possible value of each *Condition-variable* tag. The columns contain 1's for rows corresponding to rows (e.g., events) for which that condition applies and 0's otherwise.

A JSON file with a single *factor_hed_type* transformation operation.

```
[{
    "operation": "factor_hed_type"
    "description": "Factor based on the sex of the images being presented."
    "parameters": {
        "type_tag": "Condition-variable",
        "type_values": []
    }
}]
```

The results of executing this *factor_hed-tags* operation on the ***sample remodel event file*** using the ***sample remodel sidecar file*** for HED annotations are:

Results of *factor_hed_type*.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex	Image-sex.Female-image-cond	Image-sex.Male-image-cond
0.0776	0.5083	go	n/a	0.565	correct	right	female	1	0
5.5774	0.5083	unsuccesful_stop	0.2	0.49	correct	right	female	1	0
9.5856	0.5084	go	n/a	0.45	correct	right	female	1	0
13.5939	0.5083	succesful_stop	0.2	n/a	n/a	n/a	female	1	0
17.1021	0.5083	unsuccesful_stop	0.25	0.633	correct	left	male	0	1
21.6103	0.5083	go	n/a	0.443	correct	left	male	0	1

6.14.8.4 Merge consecutive

Sometimes a single long event in experimental logs is represented by multiple repeat events. The *merge_consecutive* operation collapses these consecutive repeat events into one event with duration updated to encompass the temporal extent of the merged events.

Merge consecutive parameters

Parameters for the *merge_consecutive* operation.

Parameter	Type	Description
column_name	str	The name of the column which is the basis of the merge.
event_code	str, int, float	The value in *column_name* that triggers the merge.
match_columns	list	Columns whose values must match to collapse events.
set_durations	bool	If true, set durations based on merged events.
ignore_missing	bool	If true, missing *column_name* or *match_columns* do not raise an error.

The first of the group of rows (each representing an event) to be merged is called the anchor for the merge. After the merge, it is the only row in the group that remains in the data file. The result is identical to its original version, except for the value in the `duration` column.

If the *set_duration* parameter is true, the new duration is calculated as though the event began with the onset of the first event (the anchor row) in the group and ended at the point where all the events in the group have ended. This method allows for small gaps between events and for events in which an intermediate event in the group ends after later events. If the *set_duration* parameter is false, the duration of the merged row is set to `n/a`.

If the data file has other columns besides `onset`, `duration` and column *column_name*, the values in the other columns must be considered during the merging process. The *match_columns* is a list of the other columns whose values must agree with those of the anchor row in order for a merge to occur. If *match_columns* is empty, the other columns in each row are not taken into account during the merge.

Merge consecutive example

The *merge_consecutive* operation in the following example causes consecutive `succesful_stop` events whose `stop_signal_delay`, `response_hand`, and `sex` columns have the same values to be merged into a single event.

A JSON file with a single *merge_consecutive* transformation operation.

```
[{
    "operation": "merge_consecutive"
    "description": "Merge consecutive *succesful_stop* events that match the *match_
↪columns."
    "parameters": {
        "column_name": "trial_type",
        "event_code": "succesful_stop",
        "match_columns": ["stop_signal_delay", "response_hand", "sex"],
        "set_durations": true,
        "ignore_missing": true
    }
}]
```

When this operation is applied to the following input file, the three events with a value of `succesful_stop` in the `trial_type` column starting at `onset` value 13.5939 are merged into a single event.

Input file for a *merge_consecutive* operation.

onset	duration	trial_type	stop_signal_delay	response_hand	sex
0.0776	0.5083	go	n/a	right	female
5.5774	0.5083	unsuccesful_stop	0.2	right	female
9.5856	0.5084	go	n/a	right	female
13.5939	0.5083	succesful_stop	0.2	n/a	female
14.2	0.5083	succesful_stop	0.2	n/a	female
15.3	0.7083	succesful_stop	0.2	n/a	female
17.3	0.5083	unsuccesful_stop	0.25	n/a	female
19.0	0.5083	unsuccesful_stop	0.25	n/a	female
21.1021	0.5083	unsuccesful_stop	0.25	left	male
22.6103	0.5083	go	n/a	left	male

Notice that the `succesful_stop` event at `onset` value `17.3` is not merged because the `stop_signal_delay` column value does not match the value in the previous event. The final result has `duration` computed as `2.4144` = `15.3` + `0.7083` - `13.5939`.

The results of the *merge_consecutive* operation.

onset	duration	trial_type	stop_signal_delay	response_hand	sex
0.0776	0.5083	go	n/a	right	female
5.5774	0.5083	unsuccesful_stop	0.2	right	female
9.5856	0.5084	go	n/a	right	female
13.5939	2.4144	succesful_stop	0.2	n/a	female
17.3	2.2083	unsuccesful_stop	0.25	n/a	female
21.1021	0.5083	unsuccesful_stop	0.25	left	male
22.6103	0.5083	go	n/a	left	male

The events that had onsets at `17.3` and `19.0` are also merged in this example

6.14.8.5 Remap columns

The *remap_columns* operation maps combinations of values in *m* specified columns of a data file into values in *n* columns using a defined mapping. Remapping is useful during analysis to create columns in event files that are more directly useful or informative for a particular analysis.

Remapping is also important during the initial generation of event files from experimental logs. The log files generated by experimental control software often generate a code for each type of log entry. Remapping can be used to convert the column containing these codes into one or more columns with more informative information.

Remap columns parameters

Parameters for the *remap_columns* operation.

Parameter	Type	Description
source_columns	list	A list of *m* names of the source columns for the map.
destination_columns	list	A list of *n* names of the destination columns for the map.
map_list	list	A list of mappings. Each element is a list of *m* source column values followed by *n* destination values. Mapping source values are treated as strings.
ignore_missing	bool	If false, source column values not in the map generate "n/a" destination values instead of errors.
integer_sources	list	[**Optional**] A list of source columns that are integers. The *integer_sources* must be a subset of *source_columns*.

A column cannot be both a source and a destination, and all source columns must be present in the data files. New columns are created for destination columns that are missing from a data file.

The *remap_columns* operation only works for columns containing strings or integers, as it is meant for remapping categorical codes. You must specify the which source columns contain integers so that `n/a` values can be handled appropriately.

The *map_list* parameter specifies how each unique combination of values from the source columns will be mapped into the destination columns. If there are *m* source columns and *n* destination columns, then each entry in *map_list* must be a list with $m + n$ elements. The first *m* elements are the key values from the source columns. The *map_list* should have targets for all combinations of values that appear in the *m* source columns unless *ignore_missing* is true.

After remapping, the tabular file will contain both source and destination columns. If you wish to replace the source columns with the destination columns, use a *remove_columns* transformation after the *remap_columns*.

Remap columns example

The *remap_columns* operation in the following example creates a new column called *response_type* based on the unique values in the combination of columns *response_accuracy* and *response_hand*.

A JSON file with a single *remap_columns* transformation operation.

```
[{
    "operation": "remap_columns",
    "description": "Map response_accuracy and response hand into a single column.",
    "parameters": {
        "source_columns": ["response_accuracy", "response_hand"],
        "destination_columns": ["response_type"],
        "map_list": [["correct", "left", "correct_left"],
                     ["correct", "right", "correct_right"],
                     ["incorrect", "left", "incorrect_left"],
                     ["incorrect", "right", "incorrect_left"],
                     ["n/a", "n/a", "n/a"]],
        "ignore_missing": true
    }
}]
```

In this example there are two source columns and one destination column, so each entry in *map_list* must be a list with three elements two source values and one destination value). Since all the values in *map_list* are strings, the optional *integer_sources* list is not needed.

The results of executing the previous *remap_column* command on the ***sample remodel event file*** are:

Mapping columns *response_accuracy* and *response_hand* into a *response_type* column.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex	response_type
0.0776	0.5083	go	n/a	0.565	correct	right	female	correct_right
5.5774	0.5083	unsuccessful_stop	0.2	0.49	correct	right	female	correct_right
9.5856	0.5084	go	n/a	0.45	correct	right	female	correct_right
13.5939	0.5083	successful_stop	0.2	n/a	n/a	n/a	female	n/a
17.1021	0.5083	unsuccessful_stop	0.25	0.633	correct	left	male	correct_left
21.6103	0.5083	go	n/a	0.443	correct	left	male	correct_left

In this example, *remap_columns* combines the values from columns `response_accuracy` and `response_hand` to produce a new column called `response_type` that specifies both response hand and correctness information using a single code.

6.14.8.6 Remove columns

Sometimes columns are added during intermediate processing steps. The *remove_columns* operation is useful for cleaning up unnecessary columns after these processing steps complete.

Remove columns parameters

Parameters for the *remove_columns* operation.

Parameter	Type	Description
column_names	list of str	A list of columns to remove.
ignore_missing	boolean	If true, missing columns are ignored, otherwise raise `KeyError`.

If one of the specified columns is not in the file and the *ignore_missing* parameter is *false*, a `KeyError` is raised for the missing column.

Remove columns example

The following example specifies that the *remove_columns* operation should remove the `stop_signal_delay`, `response_accuracy`, and `face` columns from the tabular data.

A JSON file with a single *remove_columns* transformation operation.

```
[{
    "operation": "remove_columns",
    "description": "Remove extra columns before the next step.",
    "parameters": {
        "column_names": ["stop_signal_delay", "response_accuracy", "face"],
        "ignore_missing": true
    }
}]
```

The results of executing this operation on the ***sample remodel event file*** are shown below. The *face* column is not in the data, but it is ignored, since *ignore_missing* is true. If *ignore_missing* had been false, a `KeyError` would have been raised.

Results of executing the *remove_columns*.

onset	duration	trial_type	response_time	response_hand	sex
0.0776	0.5083	go	0.565	right	female
5.5774	0.5083	unsuccesful_stop	0.49	right	female
9.5856	0.5084	go	0.45	right	female
13.5939	0.5083	succesful_stop	n/a	n/a	female
17.1021	0.5083	unsuccesful_stop	0.633	left	male
21.6103	0.5083	go	0.443	left	male

6.14.8.7 Remove rows

The *remove_rows* operation eliminates rows in which the named column has one of the specified values. This operation is useful for removing event markers corresponding to particular types of events or, for example having `n/a` in a particular column.

Remove rows parameters

Parameters for *remove_rows*.

Parameter	Type	Description
column_name	str	The name of the column to be tested.
remove_values	list	A list of values to be tested for removal.

The operation does not raise an error if a data file does not have a column named *column_name* or is missing a value in *remove_values*.

Remove rows example

The following *remove_rows* operation removes the rows whose *trial_type* column contains either `succesful_stop` or `unsuccesful_stop`.

A JSON file with a single *remove_rows* transformation operation.

```
[{
    "operation": "remove_rows",
    "description": "Remove rows where trial_type is either succesful_stop or unsuccesful_
↪stop.",
    "parameters": {
        "column_name": "trial_type",
        "remove_values": ["succesful_stop", "unsuccesful_stop"]
    }
}]
```

The results of executing the previous *remove_rows* operation on the ***sample remodel event file*** are:

The results of executing the previous *remove_rows* operation.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex
0.0776	0.5083	go	n/a	0.565	correct	right	female
9.5856	0.5084	go	n/a	0.45	correct	right	female
21.6103	0.5083	go	n/a	0.443	correct	left	male

After removing rows with `trial_type` equal to `succesful_stop` or `unsuccesful_stop` only the three `go` trials remain.

6.14.8.8 Rename columns

The `rename_columns` operations uses a dictionary to map old column names into new ones.

Rename columns parameters

Parameters for *rename_columns*.

Parameter	Type	Description
column_mapping	dict	The keys are the old column names and the values are the new names.
ignore_missing	bool	If false, a `KeyError` is raised if a dictionary key is not a column name.

If *ignore_missing* is false, a `KeyError` is raised if a column specified in the mapping does not correspond to a column name in the data file.

Rename columns example

The following example renames the `stop_signal_delay` column to be `stop_delay` and the `response_hand` to be `hand_used`.

A JSON file with a single *rename_columns* transformation operation.

```
[{
    "operation": "rename_columns",
    "description": "Rename columns to be more descriptive.",
    "parameters": {
        "column_mapping": {
            "stop_signal_delay": "stop_delay",
            "response_hand": "hand_used"
        },
        "ignore_missing": true
    }
}]
```

The results of executing the previous *rename_columns* operation on the ***sample remodel event file*** are:

After the *rename_columns* operation is executed, the sample events file is:

onset	duration	trial_type	stop_delay	response_time	response_accuracy	hand_used	sex
0.0776	0.5083	go	n/a	0.565	correct	right	female
5.5774	0.5083	unsuccesful_stop	0.2	0.49	correct	right	female
9.5856	0.5084	go	n/a	0.45	correct	right	female
13.5939	0.5083	succesful_stop	0.2	n/a	n/a	n/a	female
17.1021	0.5083	unsuccesful_stop	0.25	0.633	correct	left	male
21.6103	0.5083	go	n/a	0.443	correct	left	male

6.14.8.9 Reorder columns

The *reorder_columns* operation reorders the indicated columns in the specified order. This operation is often used to place the most important columns near the beginning of the file for readability or to assure that all the data files in dataset have the same column order. Additional parameters control how non-specified columns are treated.

Reorder columns parameters

Parameters for the *reorder_columns* operation.

Parameter	Type	Description
column_order	list	A list of columns in the order they should appear in the data.
ignore_missing	bool	Controls handling column names in the reorder list that aren't in the data.
keep_others	bool	Controls handling of columns not in the reorder list.

If *ignore_missing* is true and items in the reorder list do not exist in the file, the missing columns are ignored. On the other hand, if *ignore_missing* is false, a column name in the reorder list that is missing from the data raises a *ValueError*.

The *keep_others* parameter controls whether columns in the data that do not appear in the *column_order* list are dropped (*keep_others* is false) or put at the end in the relative order that they appear in the file (*keep_others* is true).

BIDS event files are required to have `onset` and `duration` as the first and second columns, respectively.

Reorder columns example

The *reorder_columns* operation in the following example specifies that the first four columns of the dataset should be: `onset`, `duration`, `response_time`, and `trial_type`. Since *keep_others* is false, these will be the only columns retained.

A JSON file with a single *reorder_columns* transformation operation.

```
[{
    "operation": "reorder_columns",
    "description": "Reorder columns.",
    "parameters": {
        "column_order": ["onset", "duration", "response_time",  "trial_type"],
        "ignore_missing": true,
        "keep_others": false
    }
}]
```

The results of executing the previous *reorder_columns* transformation on the ***sample remodel event file*** are:

Results of *reorder_columns*.

onset	duration	response_time	trial_type
0.0776	0.5083	0.565	go
5.5774	0.5083	0.49	unsuccesful_stop
9.5856	0.5084	0.45	go
13.5939	0.5083	n/a	succesful_stop
17.1021	0.5083	0.633	unsuccesful_stop
21.6103	0.5083	0.443	go

6.14.8.10 Split rows

The *split_rows* operation is often used to convert event files from trial-level encoding to event-level encoding.

In **trial-level** encoding, all the events in a single trial (usually some variation of the cue-stimulus-response-feedback-ready sequence) are represented by a single row in the data file. Often, the onset corresponds to the presentation of the stimulus, and the other events are not reported or are implicitly reported.

In **event-level** encoding, each row represents the temporal marker for a single event. In this case a trial consists of a sequence of multiple events.

Split rows parameters

Parameters for the *split_rows* operation.

Parameter	Type	Description
anchor_column	str	The name of the column that will be used for split_rows codes.
new_events	dict	Dictionary whose keys are the codes to be inserted as new eventsin the *anchor_column* and whose values are dictionaries withkeys *onset_source*, *duration*, and *copy_columns*.
remove_parent_event	bool	If true, remove parent event.

The *split_rows* operation requires an *anchor_column*, which could be an existing column or a new column to be appended to the data. The purpose of the *anchor_column* is to hold the codes for the new events.

The *new_events* dictionary has the new events to be created. The keys are the new event codes to be inserted into the *anchor_column*. The values in *new_events* are themselves dictionaries. Each of these dictionaries has three keys:

- *onset_source* is a list of items to be added to the *onset* of the event row being split to produce the `onset` column value for the new event. These items can be any combination of numerical values and column names.
- *duration* a list of numerical values and/or column names whose values are to be added to compute the `duration` column value for the new event.
- *copy_columns* a list of column names whose values should be copied into each new event. Unlisted columns are filled with `n/a`.

The *split_rows* operation sorts the split rows by the `onset` column and raises a `TypeError` if the `onset` and `duration` are improperly defined. The `onset` column is converted to numeric values as part splitting process.

Split rows example

The *split_rows* operation in the following example specifies that new rows should be added to encode the response and stop signal. The anchor column is `trial_type`.

A JSON file with a single *split_rows* transformation operation.

```
[{
  "operation": "split_rows",
  "description": "add response events to the trials.",
        "parameters": {
            "anchor_column": "trial_type",
            "new_events": {
                "response": {
                    "onset_source": ["response_time"],
                    "duration": [0],
                    "copy_columns": ["response_accuracy", "response_hand", "sex", "trial_
↪number"]
                },
                "stop_signal": {
                    "onset_source": ["stop_signal_delay"],
                    "duration": [0.5],
                    "copy_columns": ["trial_number"]
                }
            },
            "remove_parent_event": false
        }
    }]
```

The results of executing this *split_rows* operation on the ***sample remodel event file*** are:

Results of the previous *split_rows* operation.

onset	duration	trial_type	stop_signal_delay	response_time	response_accuracy	response_hand	sex
0.0776	0.5083	go	n/a	0.565	correct	right	female
0.6426	0	response	n/a	n/a	correct	right	female
5.5774	0.5083	unsuccessful_stop	0.2	0.49	correct	right	female
5.7774	0.5	stop_signal	n/a	n/a	n/a	n/a	n/a
6.0674	0	response	n/a	n/a	correct	right	female
9.5856	0.5084	go	n/a	0.45	correct	right	female
10.0356	0	response	n/a	n/a	correct	right	female
13.5939	0.5083	successful_stop	0.2	n/a	n/a	n/a	female
13.7939	0.5	stop_signal	n/a	n/a	n/a	n/a	n/a
17.1021	0.5083	unsuccessful_stop	0.25	0.633	correct	left	male
17.3521	0.5	stop_signal	n/a	n/a	n/a	n/a	n/a
17.7351	0	response	n/a	n/a	correct	left	male
21.6103	0.5083	go	n/a	0.443	correct	left	male
22.0533	0	response	n/a	n/a	correct	left	male

In a full processing example, it might make sense to rename `trial_type` to be `event_type` and to delete the `response_time` and the `stop_signal_delay` columns, since these items have been unfolded into separate events. This could be accomplished in subsequent clean-up operations.

6.14.9 Remodel summarizations

Summarizations differ transformations in two respects: they do not modify the input data file, and they keep information about the results from each file that has been processed. Summarization operations may be used at several points in the operation list as checkpoints during debugging as well as for their more typical informational uses.

All summary operations have two required parameters: *summary_name* and *summary_filename*.

The *summary_name* is the unique key used to identify the particular incarnation of this summary in the dispatcher. Care should be taken to make sure that the *summary_name* is unique within a given JSON remodeling file if the same summary operation is used more than once within the file (e.g. for before and after summary information).

The *summary_filename* should also be unique and is used for saving the summary upon request. When the remodeler is applied to full datasets rather than single files, the summaries are saved in the `derivatives/remodel/summaries` directory under the dataset root. A time stamp and file extension are appended to the *summary_filename* when the summary is saved.

6.14.9.1 Summarize column names

The *summarize_column_names* tracks the unique column name patterns found in data files across the dataset and which files have these column names. This summary is useful for determining whether there are any non-conforming data files.

Often event files associated with different tasks have different column names, and this summary can be used to verify that the files corresponding to the same task have the same column names.

A more problematic issue is when some event files for the same task have reordered column names or use different column names.

Summarize column names parameters

The *summarize_column_names* operation has no parameters and only requires the *summary_name* and the *summary_filename* to specify the operation.

The *summarize_column_names* operation only has the two parameters required of all summaries.

Parameters for the *summarize_column_names* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

Summarize column names example

The following example remodeling file produces a summary, which when saved will appear with file name `AOMIC_column_names_xxx.txt` or `AOMIC_column_names_xxx.json` where `xxx` is a timestamp.

A JSON file with a single *summarize_column_names* summarization operation.

```
[{
    "operation": "summarize_column_names",
    "description": "Summarize column names.",
    "parameters": {
        "summary_name": "AOMIC_column_names",
        "summary_filename": "AOMIC_column_names"
    }
}]
```

When this operation is applied to the *sample remodel event file*, the following text summary is produced.

Result of applying *summarize_column_names* to the sample remodel file.

```
Summary name: AOMIC_column_names
Summary type: column_names
```

(continues on next page)

(continued from previous page)

```
Summary filename: AOMIC_column_names

Summary details:

Dataset: Number of files=1
    Columns: ['onset', 'duration', 'trial_type', 'stop_signal_delay', 'response_time',
↪'response_accuracy', 'response_hand', 'sex']
        sub-0013_task-stopsignal_acq-seq_events.tsv

Individual files:

sub-0013_task-stopsignal_acq-seq_events.tsv:
   ['onset', 'duration', 'trial_type', 'stop_signal_delay', 'response_time', 'response_
↪accuracy', 'response_hand', 'sex']
```

Since we are only summarizing one event file, there is only one unique pattern – corresponding to the columns: *onset*, *duration*, *trial_type*, *stop_signal_delay*, *response_time*, *response_accuracy*, *response_hand*, and *response_time*.

When the dataset has multiple column name patterns, the summary lists unique pattern separately along with the names of the data files that have this pattern.

The JSON version of the summary is useful for programmatic manipulation, while the text version shown above is more readable.

6.14.9.2 Summarize column values

The summarize column values operation provides a summary of the number of times various column values appear in event files across the dataset.

Summarize column values parameters

The following table lists the parameters required for using the summary.

Parameters for the *summarize_column_values* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
skip_columns	list	A list of column names to omit from the summary.
value_columns	list	A list of columns to omit the listing unique values.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

In addition to the standard parameters, *summary_name* and *summary_filename* required of all summaries, the *summarize_column_values* operation requires two additional lists to be supplied. The *skip_columns* list specifies the names of columns to skip entirely in the summary. Typically, the `onset`, `duration`, and `sample` columns are skipped, since they have unique values for each row and their values have limited information.

The *summarize_column_values* is mainly meant for creating summary information about columns containing a finite number of distinct values. Columns that contain numeric information will usually have distinct entries for each row in a tabular file and are not amenable to such summarization. These columns could be specified as *skip_columns*, but another option is to designate them as *value_columns*. The *value_columns* are reported in the summary, but their distinct values are not reported individually.

For datasets that include multiple tasks, the event values for each task may be distinct. The *summarize_column_values* operation does not separate by task, but expects the calling programs filter the files by task as desired. The `run_remodel` program supports selecting files corresponding to a particular task.

Summarize column values example

The following example shows the JSON for including this operation in a remodeling file.

A JSON file with a single *summarize_column_values* summarization operation.

```json
[{
   "operation": "summarize_column_values",
   "description": "Summarize the column values in an excerpt.",
   "parameters": {
       "summary_name": "AOMIC_column_values",
       "summary_filename": "AOMIC_column_values",
       "skip_columns": ["onset", "duration"],
       "value_columns": ["response_time", "stop_signal_delay"]
   }
}]
```

A text format summary of the results of executing this operation on the ***sample remodel event file*** is shown in the following example.

Sample *summarize_column_values* operation results in text format.

```
Summary name: AOMIC_column_values
Summary type: column_values
Summary filename: AOMIC_column_values

Overall summary:
Dataset: Total events=6 Total files=1
   Categorical column values[Events, Files]:
      response_accuracy:
         correct[5, 1] n/a[1, 1]
      response_hand:
         left[2, 1] n/a[1, 1] right[3, 1]
      sex:
         female[4, 1] male[2, 1]
      trial_type:
         go[3, 1] succesful_stop[1, 1] unsuccesful_stop[2, 1]
   Value columns[Events, Files]:
      response_time[6, 1]
      stop_signal_delay[6, 1]
```

(continues on next page)

(continued from previous page)

```
Individual files:

sub-0013_task-stopsignal_acq-seq_events.tsv:
Total events=200
      Categorical column values[Events, Files]:
         response_accuracy:
            correct[5, 1] n/a[1, 1]
         response_hand:
            left[2, 1] n/a[1, 1] right[3, 1]
         sex:
            female[4, 1] male[2, 1]
         trial_type:
            go[3, 1] succesful_stop[1, 1] unsuccesful_stop[2, 1]
      Value columns[Events, Files]:
         response_time[6, 1]
         stop_signal_delay[6, 1]
```

Because the ***sample remodel event file*** only has 6 events, we expect that no value will be represented in more than 6 events. The column names corresponding to value columns just have the event counts in them.

This command was executed with the `-i` option in `run_remodel`, results from the individual data files are shown after the overall summary. The individual results are similar to the overall summary because only one data file was processed.

For a more extensive example see the **text** and **JSON** format summaries of the sample dataset **ds003645s_hed** using the **summarize_columns_rmdl.json** remodeling file.

6.14.9.3 Summarize definitions

The summarize definitions operation provides a summary of the `Def-expand` tags found across the dataset, nothing any ambiguous or erroneous ones. If working on a BIDS dataset, it will initialize with the known definitions from the sidecar, reporting any deviations from the known definitions as errors.

Summarize definitions parameters

The following table lists the parameters required for using the summary.

Parameters for the *summarize_definitions* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

The *summarize_definitions* is mainly meant for verifying consistency in unknown `Def-expand` tags. This comes up where you have an assembled dataset, but no longer have the definitions stored (or never created them to begin with).

Summarize definitions example

The following example shows the JSON for including this operation in a remodeling file.

A JSON file with a single *summarize_definitions* summarization operation.

```
[{
   "operation": "summarize_definitions",
   "description": "Summarize the definitions used in this dataset.",
   "parameters": {
       "summary_name": "HED_column_definition_summary",
       "summary_filename": "HED_column_definition_summary"
   }
}]
```

A text format summary of the results of executing this operation on the **sub-003_task-FacePerception_run-3_events.tsv** file of the eeg_ds_003645s_hed_column dataset is shown in the following example.

Sample *summarize_definitions* operation results in text format.

```
Summary name: HED_column_definition_summary
Summary type: definitions
Summary filename: HED_column_definition_summary

Overall summary:
   Known Definitions: 17 items
      cross-only: 2 items
         description: A white fixation cross on a black background in the center of the␣
↪screen.
         contents: (Visual-presentation,(Background-view,Black),(Foreground-view,(Center-
↪of,Computer-screen),(Cross,White)))
      face-image: 2 items
         description: A happy or neutral face in frontal or three-quarters frontal pose␣
↪with long hair cropped presented as an achromatic foreground image on a black␣
↪background with a white fixation cross superposed.
         contents: (Visual-presentation,(Background-view,Black),(Foreground-view,
↪((Center-of,Computer-screen),(Cross,White)),(Grayscale,(Face,Hair,Image))))
      circle-only: 2 items
         description: A white circle on a black background in the center of the screen.
         contents: (Visual-presentation,(Background-view,Black),(Foreground-view,
↪((Center-of,Computer-screen),(Circle,White))))
      press-left-finger: 2 items
         description: The participant presses a key with the left index finger to␣
↪indicate a face symmetry judgment.
         contents: ((Index-finger,(Experiment-participant,Left-side-of)),(Keyboard-key,
↪Press))
      press-right-finger: 2 items
         description: The participant presses a key with the right index finger to␣
↪indicate a face symmetry evaluation.
         contents: ((Index-finger,(Experiment-participant,Right-side-of)),(Keyboard-key,
↪Press))
```

(continues on next page)

(continued from previous page)

```
      famous-face-cond: 2 items
         description: A face that should be recognized by the participants
         contents: (Condition-variable/Face-type,(Image,(Face,Famous)))
      unfamiliar-face-cond: 2 items
         description: A face that should not be recognized by the participants.
         contents: (Condition-variable/Face-type,(Image,(Face,Unfamiliar)))
      scrambled-face-cond: 2 items
         description: A scrambled face image generated by taking face 2D FFT.
         contents: (Condition-variable/Face-type,(Image,(Disordered,Face)))
      first-show-cond: 2 items
         description: Factor level indicating the first display of this face.
         contents: ((Condition-variable/Repetition-type,Item-interval/0,(Face,Item-count/
↪1)))
      immediate-repeat-cond: 2 items
         description: Factor level indicating this face was the same as previous one.
         contents: ((Condition-variable/Repetition-type,Item-interval/1,(Face,Item-count/
↪2)))
      delayed-repeat-cond: 2 items
         description: Factor level indicating face was seen 5 to 15 trials ago.
         contents: (Condition-variable/Repetition-type,(Face,Item-count/2),(Item-
↪interval,(Greater-than-or-equal-to,Item-interval/5)))
      left-sym-cond: 2 items
         description: Left index finger key press indicates a face with above average␣
↪symmetry.
         contents: (Condition-variable/Key-assignment,((Asymmetrical,Behavioral-
↪evidence),(Index-finger,(Experiment-participant,Right-side-of))),((Behavioral-evidence,
↪Symmetrical),(Index-finger,(Experiment-participant,Left-side-of))))
      right-sym-cond: 2 items
         description: Right index finger key press indicates a face with above average␣
↪symmetry.
         contents: (Condition-variable/Key-assignment,((Asymmetrical,Behavioral-
↪evidence),(Index-finger,(Experiment-participant,Left-side-of))),((Behavioral-evidence,
↪Symmetrical),(Index-finger,(Experiment-participant,Right-side-of))))
      face-symmetry-evaluation-task: 2 items
         description: Evaluate degree of image symmetry and respond with key press␣
↪evaluation.
         contents: (Experiment-participant,Task,(Discriminate,(Face,Symmetrical)),(Face,
↪See),(Keyboard-key,Press))
      blink-inhibition-task: 2 items
         description: Do not blink while the face image is displayed.
         contents: (Experiment-participant,Inhibit-blinks,Task)
      fixation-task: 2 items
         description: Fixate on the cross at the screen center.
         contents: (Experiment-participant,Task,(Cross,Fixate))
      initialize-recording: 2 items
         description:
         contents: (Recording)
   Ambiguous Definitions: 0 items

   Errors: 0 items
```

Since this file didn't have any ambiguous or incorrect `Def-expand` groups, those sections are empty. Ambiguous definitions are those that take a placeholder, but it doesn't have enough information to be sure to which tag the placeholder applies. Erroneous ones are ones with conflicting expanded forms.

Currently, summaries are not generated for individual files, but this is likely to change in the future.

Below is a simple example showing the format when erroneous or ambiguous definitions are found.

Sample input for *summarize_definitions* operation documenting ambiguous/erroneous definitions.

```
((Def-expand/Initialize-recording,(Recording)),Onset)
((Def-expand/Initialize-recording,(Recording, Event)),Onset)
(Def-expand/Specify-age/1,(Age/1, Item-count/1))
```

Sample *summarize_definitions* operation error results in text format.

```
Summary name: HED_column_definition_summary
Summary type: definitions
Summary filename: HED_column_definition_summary

Overall summary:
   Known Definitions: 1 items
      initialize-recording: 2 items
         description:
         contents: (Recording)
   Ambiguous Definitions: 1 items
      specify-age/#: (Age/#,Item-count/#)
   Errors: 1 items
      initialize-recording:
         (Event,Recording)
```

It is assumed the first definition encountered is the correct definition, unless the first one is ambiguous. Thus, it finds (`Def-expand/Initialize-recording`,(`Recording`) and considers it valid, before encountering (`Def-expand/Initialize-recording`,(`Recording`, `Event`)), which is now deemed an error.

6.14.9.4 Summarize HED tags

The *summarize_hed_tags* operation extracts a summary of the HED tags present in the annotations of a dataset. This summary operation assumes that the structure in question is suitably annotated with HED (Hierarchical Event Descriptors). You must provide a HED schema version. If the data has annotations in a JSON sidecar, you must also provide its path.

Summarize HED tags parameters

The *summarize_hed_tags* operation has the two required parameters (*tags* and *expand_context*) in addition to the standard *summary_name* and *summary_filename* parameters.

Parameters for the *summarize_hed_tags* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
tags	dict	Dictionary with category title keys and tags in that category as values.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.
expand_context	bool	(Optional) If true, expand `Event-context` to account for onsets and offsets.

The *tags* dictionary has keys that specify how the user wishes the tags to be categorized for display. Note that these keys are titles designating display categories, not HED tags.

The *tags* dictionary values are lists of actual HED tags (or their children) that should be listed under the respective display categories.

If *expand_context* is false, the counts are calculated without expanding the event context. The option of expanding the event context to account for onset/offset effects is not implemented in this release.

The following remodeling command specifies that the tag counts should be grouped under the titles: *Sensory events*, *Agent actions*, and *Objects*. Any leftover tags will appear under the title "Other tags".

Summarize HED tags example

A JSON file with a single *summarize_hed_tags* summarization operation.

```
[{
   "operation": "summarize_hed_tags",
   "description": "Summarize the HED tags in the dataset.",
   "parameters": {
       "summary_name": "summarize_hed_tags",
       "summary_filename": "summarize_hed_tags",
       "tags": {
           "Sensory events": ["Sensory-event", "Sensory-presentation",
                              "Task-stimulus-role", "Experimental-stimulus"],
           "Agent actions": ["Agent-action", "Agent", "Action", "Agent-task-role",
                             "Task-action-type", "Participant-response"],
           "Objects": ["Item"]
         }
     }
}]
```

The results of executing this operation on the ***sample remodel event file*** are shown below.

Text summary of *summarize_hed_tags* operation on the sample remodel file.

```
Summary name: summarize_hed_tags
Summary type: hed_tag_summary
Summary filename: summarize_hed_tags

Overall summary:
Dataset: Total events=1200 Total1 file=6
        Main tags[events,files]:
                Sensory events:
                        Sensory-presentation[6,1] Visual-presentation[6,1] Auditory-
↪presentation[3,1]
                Agent actions:
                        Incorrect-action[2,1] Correct-action[1,1]
                Objects:
                        Image[6,1]
        Other tags[events,files]:
                Label[6,1] Def[6,1] Delay[3,1]

Individual files:

aomic_sub-0013_excerpt_events.tsv:
Total events=6
   Main tags[events,files]:
       Sensory events:
          Sensory-presentation[6,1] Visual-presentation[6,1] Auditory-presentation[3,1]
       Agent actions:
          Incorrect-action[2,1] Correct-action[1,1]
       Objects:
          Image[6,1]
   Other tags[events,files]:
       Label[6,1] Def[6,1] Delay[3,1]
```

The HED tag *Task-action-type* was specified in the "Agent actions" category, *Incorrect-action* and *Correct-action*, which are children of *Task-action-type* in the **HED schema**, will appear with counts in the list under this category.

The sample events file had 6 events, including 1 correct action and 2 incorrect actions. Since only one file was processed, the information for *Dataset* was similar to that presented under *Individual files*.

For a more extensive example, see the **text** and **JSON** format summaries of the sample dataset **ds003645s_hed** using the **summarize_hed_tags_rmdl.json** remodeling file.

6.14.9.5 Summarize HED type

The *summarize_hed_type* operation is designed to extract experimental design matrices or other experimental structure. This summary operation assumes that the structure in question is suitably annotated with HED (Hierarchical Event Descriptors). The **HED conditions and design matrices** explains how this works.

Summarize HED type parameters

The *summarize_hed_type* operation provides detailed information about a specified tag, usually `Condition-variable` or `Task`. This summary provides useful information about experimental design.

Parameters for the *summarize_hed_type* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
type_tag	str	Tag to produce a summary for (most often *condition-variable*).
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

In addition to the two standard parameters (*summary_name* and *summary_filename*), the *type_tag* parameter is required. Only one tag can be given, so you must provide a separate operations in the remodel file for multiple type tags.

Summarize HED type example

A JSON file with a single *summarize_hed_type* summarization operation.

```
[{
   "operation": "summarize_hed_type",
   "description": "Summarize column names.",
   "parameters": {
       "summary_name": "AOMIC_condition_variables",
       "summary_filename": "AOMIC_condition_variables",
       "type_tag": "condition-variable"
   }
}]
```

The results of executing this operation on the ***sample remodel event file*** are shown below.

Text summary of *summarize_hed_types* operation on the sample remodel file.

```
Summary name: AOMIC_condition_variables
Summary type: hed_type_summary
Summary filename: AOMIC_condition_variables

Overall summary:

Dataset: Type=condition-variable Type values=1 Total events=6 Total files=1
```

(continues on next page)

(continued from previous page)

```
   image-sex: 2 levels in 6 event(s)s out of 6 total events in 1 file(s)
       female-image-cond [4,1]: ['Female', 'Image', 'Face']
       male-image-cond [2,1]: ['Male', 'Image', 'Face']

Individual files:

aomic_sub-0013_excerpt_events.tsv:
Type=condition-variable Total events=6
      image-sex: 2 levels in 6 events
         female-image-cond [4 events, 1 files]:
            Tags: ['Female', 'Image', 'Face']
         male-image-cond [2 events, 1 files]:
            Tags: ['Male', 'Image', 'Face']
```

Because *summarize_hed_type* is a HED operation, a HED schema version is required and a JSON sidecar is also usually needed. This summary was produced by using `hed_version="8.1.0"` when creating the `dispatcher` and using the ***sample remodel sidecar file*** in the `do_op`. The sidecar provides the annotations that use the `condition-variable` tag in the summary.

For a more extensive example, see the **text** and **JSON** format summaries of the sample dataset **ds003645s_hed** using the **summarize_hed_types_rmdl.json** remodeling file.

6.14.9.6 Summarize HED validation

The *summarize_hed_validation* operation runs the HED validator on the requested data and produces a summary of the errors. See the ***HED validation guide*** for available methods of running the HED validator.

Summarize HED validation parameters

In addition to the required *summary_name* and *summary_filename* parameters, the *summarize_hed_validation* operation has a required boolean parameter *check_for_warnings*. If *check_for_warnings* is false, the summary will not report warnings.

Parameters for the *summarize_hed_validation* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
check_for_warnings	bool	If true, warnings are reported, otherwise warnings are ignored.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

The *summarize_hed_validation* is a HED operation and the calling program must provide a HED schema version and usually a JSON sidecar containing the HED annotations.

The validation process takes place in two stages: first the JSON sidecar is validated. This strategy is used because a single error in the JSON sidecar can generate an error message for every line in the corresponding data file.

If the JSON sidecar has errors (warnings don't count), the validation process is terminated without validation of the data file and assembled HED annotations.

If the JSON sidecar does not have errors, the validator assembles the annotations for each line in the data files and validates the assembled HED annotation. Data file-wide consistency, such as matched onsets and offsets, is also checked.

Summarize HED validation example

A JSON file with a single *summarize_hed_validation* summarization operation.

```
[{
   "operation": "summarize_hed_validation",
   "description": "Summarize validation errors in the sample dataset.",
   "parameters": {
       "summary_name": "AOMIC_sample_validation",
       "summary_filename": "AOMIC_sample_validation",
       "check_for_warnings": true
   }
}]
```

To demonstrate the output of the validation operation, we modified the first row of the ***sample remodel event file*** so that `trial_type` column contained the value `baloney` rather than `go`. This modification generates a warning because the meaning of `baloney` is not defined in the ***sample remodel sidecar file***. The results of executing the example operation with the modified file are shown in the following example.

Text summary of *summarize_hed_validation* operation on a modified sample data file.

```
Summary name: AOMIC_sample_validation
Summary type: hed_validation
Summary filename: AOMIC_sample_validation

Summary details:

Dataset: [1 sidecar files, 1 event files]
   task-stopsignal_acq-seq_events.json: 0 issues
   sub-0013_task-stopsignal_acq-seq_events.tsv: 6 issues

Individual files:

   sub-0013_task-stopsignal_acq-seq_events.tsv: 1 sidecar files
      task-stopsignal_acq-seq_events.json has no issues
      sub-0013_task-stopsignal_acq-seq_events.tsv issues:
            HED_UNKNOWN_COLUMN: WARNING: Column named 'onset' found in file, but not␣
↪specified as a tag column or identified in sidecars.
            HED_UNKNOWN_COLUMN: WARNING: Column named 'duration' found in file, but not␣
↪specified as a tag column or identified in sidecars.
            HED_UNKNOWN_COLUMN: WARNING: Column named 'response_time' found in file, but␣
↪not specified as a tag column or identified in sidecars.
            HED_UNKNOWN_COLUMN: WARNING: Column named 'response_accuracy' found in file,␣
↪but not specified as a tag column or identified in sidecars.
            HED_UNKNOWN_COLUMN: WARNING: Column named 'response_hand' found in file, but␣
↪not specified as a tag column or identified in sidecars.
            HED_SIDECAR_KEY_MISSING[row=0,column=2]: WARNING: Category key 'baloney'␣
```

(continues on next page)

(continued from previous page)

```
↪does not exist in column.  Valid keys are: ['succesful_stop', 'unsuccesful_stop', 'go']
```

This summary was produced using HED schema version `hed_version="8.1.0"` when creating the `dispatcher` and using the ***sample remodel sidecar file*** in the `do_op`.

6.14.9.7 Summarize sidecar from events

The summarize sidecar from events operation generates a sidecar template from the event files in the dataset.

Summarize sidecar from events parameters

The following table lists the parameters required for using the summary.

Parameters for the *summarize_sidcar_from_eventsr* operation.

Parameter	Type	Description
summary_name	str	A unique name used to identify this summary.
summary_filename	str	A unique file basename to use for saving this summary.
skip_columns	list	A list of column names to omit from the sidecar.
value_columns	list	A list of columns to treat as value columns in the sidecar.
append_timecode	bool	(Optional) If True, append a time code to filename.False is the default.

The standard summary parameters, *summary_name* and *summary_filename* are required. The *summary_name* is the unique key used to identify the particular incarnation of this summary in the dispatcher. Since a particular operation file may use a given operation multiple times, care should be taken to make sure that it is unique.

The *summary_filename* should also be unique and is used for saving the summary upon request. When the remodeler is applied to full datasets rather than single files, the summaries are saved in the `derivatives/remodel/summaries` directory under the dataset root. A time stamp and file extension are appended to the *summary_filename* when the summary is saved.

In addition to the standard parameters, *summary_name* and *summary_filename* required of all summaries, the *summarize_column_values* operation requires two additional lists to be supplied. The *skip_columns* list specifies the names of columns to skip entirely in generating the sidecar template. The *value_columns* list specifies the names of columns to treat as value columns when generating the sidecar template.

Summarize sidecar from events example

The following example shows the JSON for including this operation in a remodeling file.

A JSON file with a single *summarize_sidecar_from_events* summarization operation.

```
[{
    "operation": "summarize_sidecar_from_events",
    "description": "Generate a sidecar from the excerpted events file.",
    "parameters": {
        "summary_name": "AOMIC_generate_sidecar",
```

(continues on next page)

(continued from previous page)

```
        "summary_filename": "AOMIC_generate_sidecar",
        "skip_columns": ["onset", "duration"],
        "value_columns": ["response_time", "stop_signal_delay"]
    }
}]
```

The results of executing this operation on the ***sample remodel event file*** are shown in the following example using the text format.

Sample *summarize_sidecar_from_events* operation results in text format.

```
Summary name: AOMIC_generate_sidecar
Summary type: events_to_sidecar
Summary filename: AOMIC_generate_sidecar

Dataset: Currently no overall sidecar extraction is available

Individual files:

aomic_sub-0013_excerpt_events.tsv: Total events=6 Skip columns: ['onset', 'duration']
Sidecar:
{
    "trial_type": {
        "Description": "Description for trial_type",
        "HED": {
            "go": "(Label/trial_type, Label/go)",
            "succesful_stop": "(Label/trial_type, Label/succesful_stop)",
            "unsuccesful_stop": "(Label/trial_type, Label/unsuccesful_stop)"
        },
        "Levels": {
            "go": "Here describe column value go of column trial_type",
            "succesful_stop": "Here describe column value succesful_stop of column trial_
↪type",
            "unsuccesful_stop": "Here describe column value unsuccesful_stop of column␣
↪trial_type"
        }
    },
    "response_accuracy": {
        "Description": "Description for response_accuracy",
        "HED": {
            "correct": "(Label/response_accuracy, Label/correct)"
        },
        "Levels": {
            "correct": "Here describe column value correct of column response_accuracy"
        }
    },
    "response_hand": {
        "Description": "Description for response_hand",
        "HED": {
            "left": "(Label/response_hand, Label/left)",
```

(continues on next page)

(continued from previous page)

```
            "right": "(Label/response_hand, Label/right)"
        },
        "Levels": {
            "left": "Here describe column value left of column response_hand",
            "right": "Here describe column value right of column response_hand"
        }
    },
    "sex": {
        "Description": "Description for sex",
        "HED": {
            "female": "(Label/sex, Label/female)",
            "male": "(Label/sex, Label/male)"
        },
        "Levels": {
            "female": "Here describe column value female of column sex",
            "male": "Here describe column value male of column sex"
        }
    },
    "response_time": {
        "Description": "Description for response_time",
        "HED": "(Label/response_time, Label/#)"
    },
    "stop_signal_delay": {
        "Description": "Description for stop_signal_delay",
        "HED": "(Label/stop_signal_delay, Label/#)"
    }
}
```

The current version of the summary does not generate a dataset-wide sidecar.

6.14.10 Remodel implementation

Operations are defined as classes that extent `BaseOp` regardless of whether they are transformations or summaries. However, summaries must also implement an additional supporting class that extends `BaseSummary` to hold the summary information.

In order to be executed by the remodeling functions, an operation must appear in the `valid_operations` dictionary.

All operations must provide a `PARAMS` dictionary, a constructor that calls the base class constructor, and a `do_ops` method.

6.14.10.1 The PARAMS dictionary

The class-wide `PARAMS` dictionary has `operation`, `required_parameters` and `optional_parameters` keys. The `required_parameters` and `optional_parameters` have values that are themselves dictionaries specifying the names and types of the operation parameters.

The following example shows the `PARAMS` dictionary for the `RemoveColumnsOp` class.

The class-wide PARAMS dictionary for the RemoveColumnsOp class.

```
PARAMS = {
    "operation": "remove_columns",
    "required_parameters": {
        "column_names": list,
        "ignore_missing": bool
    },
    "optional_parameters": {}
}
```

The PARAMS dictionary allows the remodeling tools to check the syntax of the remodel input file for errors.

6.14.10.2 Operation class constructor

All the operation classes have constructors that start with a call to the superclass constructor BaseOp. The following example shows the constructor for the RemoveColumnsOp class.

The class-wide PARAMS dictionary for the RemoveColumnsOp class.

```
    def __init__(self, parameters):
        super().__init__(self.PARAMS, parameters)
        self.column_names = parameters['column_names']
        ignore_missing = parameters['ignore_missing']
        if ignore_missing:
            self.error_handling = 'ignore'
        else:
            self.error_handling = 'raise'
```

After the call to the base class constructor, the operation constructor assigns the operation-specific values to class properties and does any additional required operation-specific checks to assure that the parameters are valid.

6.14.10.3 The do_op implementation

The main method that must be implemented by each operation is do_op, which takes an instance of the Dispatcher class as the first parameter and a Pandas DataFrame representing the event file as the second parameter. A third required parameter is a name used to identify the event file in error messages and summaries. This name is usually the filename or the filepath from the dataset root. An additional optional argument, a sidecar containing HED annotations, only need be included for HED operations.

The following example shows a sample implementation for do_op.

The implementation of do_op for the RemoveColumnsOp class.

```

    def do_op(self, dispatcher, df, name, sidecar=None):
        return df.drop(self.remove_names, axis=1, errors=self.error_handling)
```

The do_op in this case is a wrapper for the underlying Pandas DataFrame operation for removing columns.

IMPORTANT NOTE: The do_op operation always assumes that n/a values have been replaced by numpy.NaN values in the incoming dataframe df. The Dispatcher class has a static method prep_data that does this replacement. At the end of running all the remodeling operations on a data file Dispatcher run_operations method replaces all of the numpy.NaN values with n/a, the value expected by BIDS. This operation is performed by the Dispatcher static method post_proc_data.

6.14.10.4 The do_op for summarization

The do_op operation for summarization operations has a slightly different form, as it serves primarily as a wrapper for the actual summary information as illustrated by the following example.

The implementation of do_op for SummarizeColumnNamesOp.

```
    def do_op(self, dispatcher, df, name, sidecar=None):
        summary = dispatcher.summary_dict.get(self.summary_name, None)
        if not summary:
            summary = ColumnNameSummary(self)
            dispatcher.summary_dict[self.summary_name] = summary
        summary.update_summary({"name": name, "column_names": list(df.columns)})
        return df
```

A do_op operation for a summarization checks the dispatcher to see if the summary name is already in the dispatcher's summary_dict. If that summary is not yet in the summary_dict, the operation creates a BaseSummary object for its summary (e.g., ColumnNameSummary) and adds this object to the dispatcher's summary_dict, otherwise the operation fetches the BaseSummary object from It then asks its BaseSummary object to update the summary based on the dataframe as explained in the next section.

6.14.10.5 Additional requirements for summarization

Any summary operation must implement a supporting class that extends BaseSummary. This class is used to hold and accumulate the information specific to the summary. This support class must implement two methods: update_summary and get_summary_details.

The update_summary method is called by its associated BaseOp operation during the do_op to update the summary information based on the current dataframe. The update_summary information takes a single parameter, which is a dictionary of information specific to this operation.

The update_summary method required to be implemented by all BaseSummary objects.

```
def update_summary(self, summary_dict)
```

In the example *do_op for ColumnNamesOp*, the dictionary is contains keys for name and `column_names.

The get_summary_details returns a dictionary with the summary-specific information currently in the summary. The BaseSummary provides universal methods for converting this summary to JSON or text format.

The get_summary_details method required to be implemented by all BaseSummary objects.

```
get_summary_details(self, verbose=True)
```

The operation associated with this instance of it associated with a given format implementation

6.15 HED Python tools

The HED (Hierarchical Event Descriptor) scripts and notebooks assume that the Python HedTools have been installed. The HedTools package is not yet available on PyPI, so you will need to install it directly from GitHub using:

```
pip install git+https://github.com/hed-standard/hed-python/@master
```

There are several types of Jupyter notebooks and other HED support tools:

- *Jupyter notebooks for HED in BIDS* - aids for HED annotation in BIDS.
- *Jupyter notebooks for data curation* - aids for summarizing and reorganizing event data.
- *Calling HED tools* - specific useful functions/classes.

6.15.1 Jupyter notebooks for HED in BIDS

The following notebooks are specifically designed to support HED annotation for BIDS datasets.

- *Summarize BIDS event files*
- *Extract a JSON sidecar template from event files*
- *Convert a JSON sidecar to a 4-column spreadsheet*
- *Validate HED in a BIDS dataset*

6.15.1.1 Summarize BIDS event files

Sometimes event files include unexpected or incorrect codes. It is a good idea to find out what is actually in the dataset event files and whether the information is consistent before starting the annotation process.

The **bids_summarize_events.ipynb** finds the dataset event files and outputs the column names and number of events for each event file. You can visually inspect the output to make sure that the event file column names are consistent across the dataset. The script also summarizes the unique values that appear in different event file columns across the dataset.

To use this notebook, substitute the specifics of your BIDS dataset for the following variables:

Variables to set in the bids_summarize_events.ipynb Jupyter notebook.

Variable	Purpose
bids_root_path	Full path to root directory of dataset.
exclude_dirs	List of directories to exclude when constructing the list of event files.
entities	Tuple of entity names used to construct a unique keys representing filenames. (See *Dictionaries of filenames* for examples of how to choose the keys.)
name_indices	Indices used to construct a unique keys representing event filenames.(See *Dictionaries of filenames* for examples of how to choose these indices.)
skip_columns	List of column names in the `events.tsv` files to skip in the analysis.

For large datasets, be sure to skip columns such as `onset` and `sample`, since the summary produces counts of the number of times each unique value appears somewhere in dataset event files.

6.15.1.2 Extract a JSON sidecar template

The usual strategy for producing machine-actionable event annotation using HED in BIDS is to create a single `events.json` sidecar file in the BIDS dataset root directory. Ideally, this sidecar will contain all the annotations needed for users to understand and analyze the data.

See the *BIDS annotation quickstart* for additional information on this strategy and an online version of the tools. The **Create a JSON template** section provides a step-by-step tutorial for using the online tool that creates a template based on the information in a single `events.tsv` file. For most datasets, this is sufficient. In contrast, the **bids_generate_sidecar.ipynb** notebook bases the extracted template on the entire dataset.

To use this notebook, substitute the specifics of your BIDS dataset for the following variables:

Variables to set in the bids_extract_sidecar.ipynb Jupyter notebook.

Variable	Purpose
bids_root_path	Full path to root directory of dataset.
exclude_dirs	List of directories to exclude when constructing the list of event files.
entities	Tuple of entity names used to construct a unique keys representing filenames. (See *Dictionaries of filenames* for examples of how to choose the keys.)
skip_columns	List of column names in the `events.tsv` files to skip in the analysis.
value_columns	List of columns names in the `events.tsv` files to annotate asas a whole rather than by individual column value.

For large datasets, be sure to skip columns such as `onset` and `sample`, since the summary produces counts of the number of times each unique value appears somewhere in dataset event files.

6.15.1.3 JSON sidecar to spreadsheet

If you have a BIDS JSON event sidecar or a sidecar template, you may find it more convenient to view and edit the HED annotations in spreadsheet rather than working with the JSON file directly as explained in the **Spreadsheet templates** tutorial.

The **bids_sidecar_to_spreadsheet.ipynb** notebook demonstrates how to extract the pertinent HED annotation to a 4-column spreadsheet (Pandas dataframe) corresponding to the HED content of a JSON sidecar. A spreadsheet representation is useful for quickly reviewing and editing HED annotations. You can easily merge the edited information back into the BIDS JSON events sidecar.

Here is an example of the spreadsheet that is produced by converting a JSON sidecar template to a spreadsheet template that is ready to edit. You should only change the values in the **description** and the **HED** columns.

Example 4-column spreadsheet template for HED annotation.

column_name	column_value	description	HED
event_type	setup_right_sym	Description for setup_right_sym	Label/setup_right_sym
event_type	show_face	Description for show_face	Label/show_face
event_type	left_press	Description for left_press	Label/left_press
event_type	show_circle	Description for show_circle	Label/show_circle
stim_file	n/a	Description for stim_file	Label/#

To use this notebook, you will need to provide the path to the JSON sidecar and a path to save the spreadsheet if you want to save it. If you don't wish to save the spreadsheet, assign `spreadsheet_filename` to be None.

The **bids_merge_sidecar.ipynb** notebook shows the complete process, from extracting the initial sidecar, to converting to a spreadsheet and then merging in another sidecar.

6.15.1.4 Validate HED in a BIDS dataset

Validating HED annotations as you develop them makes the annotation process easier and faster to debug. The **HED validation guide** discusses various HED validation issues and how to fix them.

The **bids_validate_dataset.ipynb** Jupyter notebook validates HED in a BIDS dataset using the `validate` method of `BidsDataset`. The method first gathers all the relevant JSON sidecars for each event file and validates the sidecars. It then validates the individual `events.tsv` files based on applicable sidecars.

The script requires you to set the `check_for_warnings` flag and the root path to your BIDS dataset.

Note: This validation pertains to event files and HED annotation only. It does not do a full BIDS validation.

The **bids_validate_dataset_with_libraries.ipynb** Jupyter notebook validates HED in a BIDS dataset using the `validate` method of `BidsDataset`. The example uses three schemas and also illustrates how to manually override the schema specified in `dataset_description.json` with schemas from other places. This is very useful for testing new schemas that are underdevelopment.

6.15.2 Jupyter notebooks for data curation

All data curation notebooks and other examples can now be found in the **hed-examples** repository.

6.15.2.1 Consistency of BIDS event files

Some neuroimaging modalities such as EEG, typically contain event information encoded in the data recording files, and the BIDS `events.tsv` files are generated post hoc.

In general, the following things should be checked before data is released:

1. The BIDS `events.tsv` files have the same number of events as the data recording and that onset times of corresponding events agree.
2. The associated information contained in the data recording and event files is consistent.
3. The relevant metadata is present in both versions of the data.

The example data curation scripts discussed in this section assume that two versions of each BIDS event file are present: `events.tsv` and a corresponding `events_temp.tsv` file. The example datasets that are using for these tutorials assume that the recordings are in EEG.set format. We used the runEeglabEventsToFiles MATLAB script to dump the events stored in the data.

6.15.3 Calling HED tools

This section shows examples of useful processing functions provided in HedTools:

- *Getting a list of filenames*
- *Dictionaries of filenames*
- *Logging processing steps*

6.15.3.1 Getting a list of files

Many situations require the selection of files in a directory tree based on specified criteria. The `get_file_list` function allows you to pick out files with a specified filename prefix and filename suffix and specified extensions

The following example returns a list of full paths of the files whose names end in `_events.tsv` or `_events.json` that are not in any `code` or `derivatives` directories in the `bids_root_path` directory tree. The search starts in the directory root `bids_root_path`:

Get a list of specified files in a specified directory tree.

```
file_list = get_file_list(bids_root_path, extensions=[ ".json", ".tsv"], name_suffix="_
↪events",
                          name_prefix="", exclude_dirs=[ "code", "derivatives"])
```

6.15.3.2 Dictionaries of filenames

The HED tools provide both generic and BIDS-specific classes for dictionaries of filenames.

The Many of the HED data processing tools make extensive use of dictionaries specif

BIDS-specific dictionaries of files

Files in BIDS have unique names that indicate not only what the file represents, but also where that file is located within the BIDS dataset directory tree.

BIDS file names and keys

A BIDS file name consists of an underbar-separated list of entities, each specified as a name-value pair, followed by suffix indicating the data modality.

For example, the file name `sub-001_ses-3_task-target_run-01_events.tsv` has entities subject (`sub`), task (`task`), and run (`run`). The suffix is `events` indicating that the file contains events. The extension `.tsv` gives the data format.

Modality is not the same as data format, since some modalities allow multiple formats. For example, `sub-001_ses-3_task-target_run-01_eeg.set` and `sub-001_ses-3_task-target_run-01_eeg.edf` are both acceptable representations of EEG files, but the data is in different formats.

The BIDS file dictionaries represented by the class `BidsFileDictionary` and its extension `BidsTabularDictionary` use a set combination of entities as the file key.

For a file name `sub-001_ses-3_task-target_run-01_events.tsv`, the tuple ('sub', 'task') gives a key of `sub-001_task-target`, while the tuple ('sub', 'ses', 'run) gives a key of `sub-001_ses-3_run-01`. The use of dictionaries of file names with such keys makes it easier to associate related files in the BIDS naming structure.

Notice that specifying entities ('sub', 'ses', 'run) gives the key `sub-001_ses-3_run-01` for all three files: `sub-001_ses-3_task-target_run-01_events.tsv`, `sub-001_ses-3_task-target_run-01_eeg.set` and `sub-001_ses-3_task-target_run-01_eeg.edf`. Thus, the expected usage is to create a dictionary of files of one modality.

Create a key-file dictionary for files ending in events.tsv in bids_root_path directory tree.

```
from hed.tools import FileDictionary
from hed.util import get_file_list

file_list = get_file_list(bids_root_path, extensions=[ ".set"], name_suffix="_eeg",
                          exclude_dirs=[ "code", "derivatives"])
file_dict = BidsFileDictionary(file_list, entities=('sub', 'ses', 'run) )
```

In this example, the `get_file_list` filters the files of the appropriate type, while the `BidsFileDictionary` creates a dictionary with keys such as `sub-001_ses-3_run-01` and values that are `BidsFile` objects. `BidsFile` can hold the file name of any BIDS file and keeps a parsed version of the file name.

A generic dictionary of filenames

Create a key-file dictionary for files ending in events.json in bids_root_path directory tree.

```
from hed.tools import FileDictionary
from hed.util import get_file_list

file_list = get_file_list(bids_root_path, extensions=[ ".json"], name_suffix="_events",
                          exclude_dirs=[ "code", "derivatives"])
file_dict = FileDictionary(file_list, name_indices=name_indices)
```

Keys are calculated from the filename using a `name_indices` tuple, which indicates the positions of the name-value entity pairs in the BIDS file name to use.

The BIDS filename `sub-001_ses-3_task-target_run-01_events.tsv` has three name-value entity pairs (`sub-001`, `ses-3`, `task-target`, and `run-01`) separated by underbars.

The tuple (0, 2) gives a key of `sub-001_task-target`, while the tuple (0, 3) gives a key of `sub-001_run-01`. Neither of these choices uniquely identifies the file. The tuple (0, 1, 3) gives a unique key of `sub-001_ses-3_run-01`. The tuple (0, 1, 2, 3) also works giving `sub-001_ses-3_task-target_run-01`.

If you choose the `name_indices` incorrectly, the keys for the event files will not be unique, and the notebook will throw a `HedFileError`. If this happens, modify your `name_indices` key choice to include more entity pairs.

The Jupyter notebook go_nogo_01_initial_summary.ipynb illustrates using this dictionary in a larger context.

For example, to compare the events stored in a recording file and the events in the `events.tsv` file associated with that recording, we might dump the recording events in files with the same name, but ending in `events_temp.tsv`. The `FileDictionary` class allows us to create a keyed dictionary for each of these event files.

6.15.3.3 Logging processing steps

Often event data files require considerable processing to assure internal consistency and compliance with the BIDS specification. Once this processing is done and the files have been transformed, it can be difficult to understand the relationship between the transformed files and the original data.

The HedLogger allows you to document processing steps associated with the dataset by identifying key as illustrated in the following log file excerpt:

Example output from HED logger.

```
sub-001_run-01
        Reordered BIDS columns as ['onset', 'duration', 'sample', 'trial_type',
↪'response_time', 'stim_file', 'value', 'HED']
        Dropped BIDS skip columns ['trial_type', 'value', 'response_time', 'stim_file',
↪'HED']
        Reordered EEG columns as ['sample_offset', 'event_code', 'cond_code', 'type',
↪'latency', 'urevent', 'usertags']
        Dropped EEG skip columns ['urevent', 'usertags', 'type']
        Concatenated the BIDS and EEG event files for processing
        Dropped the sample_offset and latency columns
        Saved as _events_temp1.tsv
sub-002_run-01
        Reordered BIDS columns as ['onset', 'duration', 'sample', 'trial_type',
↪'response_time', 'stim_file', 'value', 'HED']
        Dropped BIDS skip columns ['trial_type', 'value', 'response_time', 'stim_file',
↪'HED']
        Reordered EEG columns as ['sample_offset', 'event_code', 'cond_code', 'type',
↪'latency', 'urevent', 'usertags']
        Dropped EEG skip columns ['urevent', 'usertags', 'type']
        Concatenated the BIDS and EEG event files for processing
        . . .
```

Each of the lines following a key represents a print message to the logger.

The most common use for a logger is to create a file dictionary using ***make_file_dict*** and then to log each processing step using the file's key. This allows a processing step to be applied to all the relevant files in the dataset. After all the processing is complete, the print_log method outputs the logged messages by key, thus showing all the processing steps that hav been applied to each file as shown in the ***previous example***.

Using the HED logger.

```python
from hed.tools import HedLogger
status = HedLogger()
status.add(key, f"Concatenated the BIDS and EEG event files")

# ... after processing is complete output or save the log
status.print_log()
```

The HedLogger is used throughout the processing notebooks in this repository.

6.16 HED JavaScript tools

The JavaScript code for HED validation is in the validation directory of the `hed-javascript` repository located at https://github.com/hed-standard/hed-javascript.

6.16.1 Javascript tool installation

You can install the validator using `npm`:

```
npm install hed-validator
```

6.16.2 Javascript package organization

This package contains two sub-packages.

`hedValidator.validator` validates HED strings and contains the functions:

> `buildSchema` imports a HED schema and returns a JavaScript Promise object.
> `validateHedString` validates a single HED string using the returned schema object.

`hedValidator.converter` converts HED strings between short and long forms and contains the following functions:

> `buildSchema` behaves similarly to the `buildSchema` function in `hedValidator.validator` except that it does not work with attributes.
>
> `convertHedStringToShort` converts HED strings from long form to short form.
>
> `convertHedStringToLong` converts HED strings from short form to long form.

6.16.3 Javascript programmatic interface

The programmatic interface to the HED JavaScript `buildSchema` must be modified to accommodate a base HED schema and arbitrary library schemas. The BIDS validator will require additional changes to locate the relevant HED schemas from the specification given by `"HEDVersion"` in `dataset_description.json`.

The programmatic interface is similar to the JSON specification of the proposed BIDS implementation except that the `"fileName"` key has been replaced by a `"path"` key to emphasize that callers must replace filenames with full paths before calling `buildSchema`.

Example: JSON passed to buildSchema.

```
{
    "path": "/data/wonderful/code/mylocal.xml",
    "libraries": {
        "la": {
            "libraryName": "libraryA",
            "version": "1.0.2"
        },
        "lb": {
            "libraryName": "libraryB",
            "path": "/data/wonderful/code/HED_libraryB_0.5.3.xml"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

NOTE: This interface is proposed and is awaiting resolution of BIDS PR #820 on file passing to BIDS.

6.17 HED MATLAB tools

There are currently three types of support available for HED (Hierarchical Event Descriptors) supports in MATLAB:

- ***HED services in MATLAB*** - web services called from MATLAB scripts
- ***EEGLAB plug-in integration*** - EEGLAB plugins and other HED support
- ***Python HEDTools in MATLAB*** - explains how to call the HED python tools from within MATLAB.

HED services allow MATLAB programs to request the same services that are available through the browser at https://hedtools.ucsd.edu/hed.

6.17.1 HED services in MATLAB

HED RESTful services allow programs to make requests directly to the HED online tools available at https://hedtools.ucsd.edu/hed or in a locally-deployed docker module. See **HED-web** for additional information on the deployment.

The **runAllTests.m** is a main script that runs all the example code and reports whether the code runs successfully. You should start by running this script to make sure everything is working on your system, that you have Internet access, and that the HED services are available.

This script also demonstrates how to call the individual test functions. Each test function takes a host URL as a parameter and returns a list of errors. The individual test scripts illustrate how to call each type of available web service.

Target	MATLAB source	Purpose
Overall	**runAllTests.m**	Harness for running all tests.
Overall	**testGetServices.m**	List available services.
Events	**testEventServices.m**	Validation, conversion, sidecar generation.
Events	**testEventSearchServices.m**	Search, assembly.
Schema	*in progress*	For schema library developers.
Sidecars	**testSidecarServices.m**	Validation, conversion, extraction, merging.
Spreadsheets	**testSpreadsheetServices.m**	Validation, conversion.
Strings	**testStringServices.m**	Validation, conversion.

6.17.1.1 Overview of service requests

Calling HED services from MATLAB requires the following steps:

1. **Set up a session**:
    1. Establish a session by requesting a CSRF token and a cookie.
    2. Construct a header array using the token and the cookie.
2. **Create a request structure**.
3. **Make a request** using the MATLAB `webwrite`.
4. **Decode the response** returned from `webwrite`.

Usually, you will do the first step (the session setup) once at the beginning of your script to construct a fixed session header that can be used in subsequent requests in your script.

6.17.1.2 Setting up a session from MATLAB

The goal of the session setup is to construct a header that can be used in subsequent web requests. The first step is to call the getHostOptions.m. This function constructs the services URL from the host URL. The function also makes a service request to obtain a CSRF token and a cookie. The function then constructs a header and calls the MATLAB `weboptions` function to get an options object suitable for use with the MATLAB `webwrite` function use in all of our examples.

Establish a session.

```
host = 'https://hedtools.ucsd.edu/hed';
[servicesUrl, options] = getHostOptions(host)
```

The `host` should be set to the URL of the webserver that you are using. The call to `getHostOptions`, only needs to be done once at the beginning of your session. The `servicesURL` and the `options` can be used for all of your subsequent requests.

The `getHostOptions` does all the setup for using the services. As indicated by the code below, all communication is done in JSON. However, as demonstrated below, the MATLAB `webwrite` function takes a MATLAB `struct` as its `request` parameter and internally converts to the format specified in the header before making the request.

The `Timeout` parameter indicates how many seconds MATLAB will wait for a response before returning as a failed operation. The `timeout` value of 120 seconds is sufficient for most situations. However, but this can be adjusted upward or downward to suit the user. The `HeaderFields` sets the parameters of HTTP request.

Source for getHostOptions.

```
function [servicesUrl, options] = getHostOptions(host)
    csrfUrl = [host '/services'];
    servicesUrl = [host '/services_submit'];
    [cookie, csrftoken] = getSessionInfo(csrfUrl);
    header = ["Content-Type" "application/json"; ...
              "Accept" "application/json"; ...
              "X-CSRFToken" csrftoken; "Cookie" cookie];

    options = weboptions('MediaType', 'application/json', ...
                         'Timeout', 120, 'HeaderFields', header);
```

In the following examples, we assume that getHostOptions has been called to retrieve servicesUrl and options for use in the session.

6.17.1.3 Creating a request structure

The request structure is a MATLAB struct which must have a service field and can have an arbitrary number of fields depending on which service is being requested.

The simplest service is get_services, which returns a string containing information about the available services. This service requires no additional parameters.

Request a list of available HED web services.

```
request = struct('service', 'get_services');
response = webwrite(servicesUrl, request, options);
response = jsondecode(response);
```

The MATLAB webwrite returns a JSON structure as specified in the options. The MATLAB jsondecode function returns a MATLAB struct whose format is explained below in ***Decoding a service response***.

Except for get_services, all other services are of the form *target_command* where *target* is the primary type of data acted on (events, schema, sidecar, spreadsheet, or string). The possible values for *command* depend on the value of *target*. For example sidecar_validate requests that a JSON sidecar be validated.

The get_services command provides information about the HED services that are available and the parameters required. The get_services entry for sidecar_validate is the following:

The get_services entry for sidecar_validate.

```
sidecar_validate:
        Description: Validate a BIDS JSON sidecar (in string form) and return errors.
        Parameters:
                json_string
                schema_string or schema_url or schema_version
                check_for_warnings
        Returns: A list of errors if any.
```

The *Parameters* section indicates the fields in addition to the service that are needed in the request structure. For example, sidecar_validate requires a HED schema. One possibility is to read a schema into a string and provide this information in schema_string. Another possibility is to provide a URL for the schema. The most-commonly used option is to use schema_version to indicate one of the supported versions available in the **hedxml** directory of the **hed-specification** repository on GitHub.

Create a request for the sidecar_validate web service.

```
jsonText = fileread('../../../datasets/eeg_ds003645s_hed/task-FacePerception_events.json
↪');
request = struct('service', 'sidecar_validate', ...
                 'schema_version', '8.0.0', ...
```

(continues on next page)

(continued from previous page)

```
                    'json_string', jsonText, ...
                    'check_for_warnings', 'on');
```

This example reads the JSON sidecar to be validated as a character array into the variable `jsonText` and makes a request for validation using HED8.0.0.xml.

The request indicates that validation warnings as well as errors should be included in the response. If you wish to exclude warnings, use `off` instead of `on` as the `check_for_warnings` field value.

The **testSidecarServices.m** function shows complete examples of the various HED services for JSON sidecars.

6.17.1.4 Making a service request

The HED services all use the MATLAB `webwrite` to make HED web service requests. The following call uses the ***sidecar_validate request*** from the previous section.

Request the sidecar validation service.

```
response = webwrite(servicesUrl, request, options);
response = jsondecode(response);
outputReport(response, 'Example: validate a JSON sidecar');
```

The **outputReport.m** MATLAB script outputs the response in readable form with a user-provided table.

If the web server is down or times out during a request, the MATLAB `web_write` function throws an exception, and the script terminates without setting the response.

If the connection completes successfully, the response will set. The next section explains the response structure in more detail.

6.17.1.5 Decoding a service response

All HED web services return a response consisting of a JSON dictionary with 4 keys as summarized in this table.

Field name	Meaning
service	Name of the requested service.
results	Results of the operation.
error_type	Type of error if the service failed.
error_msg	Explanation of the message if the service failed.

The `jsondecode` function translates the JSON dictionary into a MATLAB structure.

The `error_type` indicates whether the service request completed successfully and was able to get results. The `error_type` **does NOT** indicate the nature of the results (for example whether a JSON sidecar was valid or not), but rather whether the server was able to complete the request without raising an exception. A failure `error_type` is highly unusual and indicates some type of unexpected internal web service error. Errors of this type should be reported using the **GitHub hed-python issues** mechanism.

The `results` structure has the actual results of the service request.

Field name	Meaning
command	Command executed in response to the service request.
command_target	Type of data on which the command was executed.
data	Data returned by the service (either processed result or a list of errors).
msg_category	Success or warning depending on the result of processing the service.
msg	Explanation of the output of the service.
output_display_name	(Optional) File name for saving return data.
schema_version	(Optional) Version of the HED schema used in the processing.

The `results` structure will always have `command`, `command_target` fields indicating what operation was performed on what type of data.

The `msg_category` will be `success` or `warning` depending on whether there were errors. The contents of the `data` field will contain different information depending on the `msg_category`. For example, if a sidecar had validation errors, `results.msg_category` will be `warning` and the `results.data` value should be interpreted as a list of errors. If the sidecar had no errors, `results.data` will be an empty string.

6.17.2 EEGLAB plug-in integration

EEGLAB is the most widely used EEG software environment for analysis of human electrophsyiological (and related) data. EEGLAB combines graphical and command-line user interfaces, making it friendly for both beginners who may who prefer a visual, and automated way of analyzing data as well as experts, who can easily customize, extend, and automate the EEGLAB tool environment by writing new EEGLAB-compatible scripts and functions.

HED is fully integrated into EEGLAB via the *HEDTools* plug-in, allowing users to annotate their EEGLAB STUDY and datasets with HED, as well as enabling HED-based data manipulation and processing.

6.17.2.1 Installing *HEDTools*

HEDTools EEGLAB plug-in can be installed using one of the following ways:

Method 1: EEGLAB Extension Manager:

Launch EEGLAB. From the main GUI select:

File –> Manage EEGLAB extension

The extension manager GUI will pop up.

From this GUI look for and select the plug-in *HEDTools* from the main window, then click into the *Install/Update* button to install the plug-in.

Method 2: Download and unzip

Download the zip file with the content of the plug-in *HEDTools* either from **HED Matlab EEGLAB plugins** or from the EEGLAB **plug-ins summary page**.

Unzip file into the folder *../eeglab/plugins* and restart the *eeglab* function in a MATLAB session.

6.17.2.2 Annotating datasets

We will start by adding HED annotations to the EEGLAB tutorial dataset.

After installing the *HEDTools* open the EEGLAB main window and load the dataset by selecting the menu item:

> **File –> Load existing dataset** .

Selecting the tutorial dataset under your eeglab installation *eeglab/sample_data/eeglab_data.set*.

Read a description of the dataset and of its included event codes by selecting:

> **Edit –> About this dataset**:

The description gives a general idea of the codes found in the event structure. Yet, inquisitive researchers interested in the nature of the stimuli (e.g., color and exact location of the squares on the screen) would have to look up the referenced paper for details.

Our goal in using HED tags is to describe the experimental events that are recorded in the *EEG.event* data structure in sufficient detail that anyone using the dataset in the future will not need to find and read a separate, detailed description of the dataset or study to understand the recorded experimental events. As demonstrated below, such annotation will allow us to extract epochs using meaningful HED tags instead of the alpha-numeric codes often associated with shared EEG data.

Launching EEGLAB HEDTools

To add and view HED tags for the dataset, from EEGLAB menu, select:

> **Edit –> Add/Edit event HED tags**.

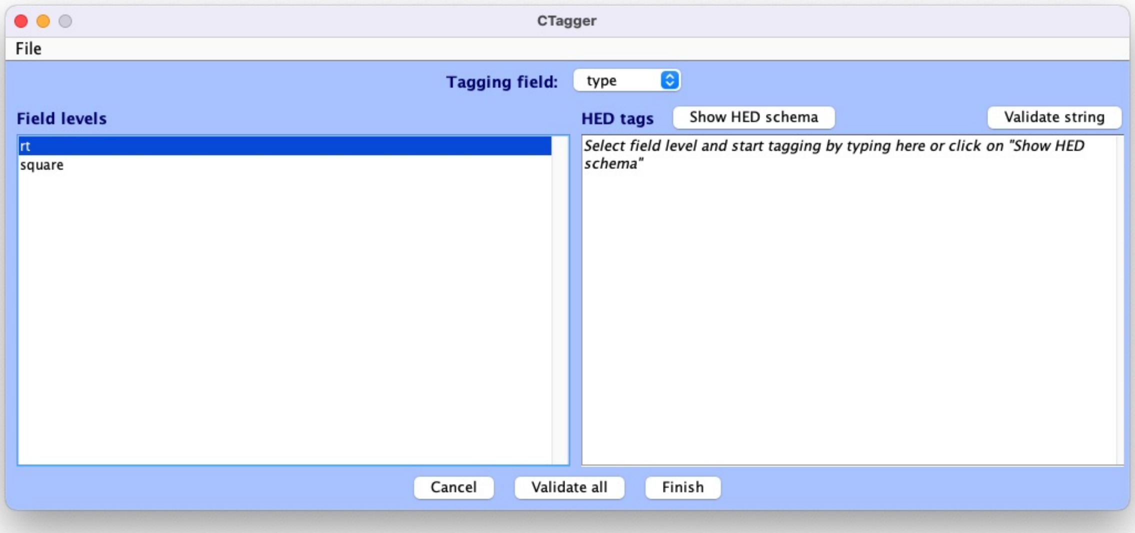
HEDTools will extract information from the *EEG.event* structure, automatically detecting the event structure fields and their unique values.

The *HEDTools* ignore the fields the event structure fields *.latency*, *.epoch*, and *.urevent*.

A window will appear asking you to verify/select categorical fields:

Here both *position* and *type* are categorical fields. *HEDTools* automatically selects fields with less than 20 unique values to be categorical, but the user can modify which values are chosen.

CTagger (for 'Community Tagger') is a graphical user interface (GUI) built to facilitate the process of adding HED tags to recorded events in existing datasets. Clicking *Continue* brings up the *CTagger* interface:



The CTagger GUI is organized using a split window strategy. The left window shows the items to be tagged, and the right window shows the current HED tags associated with the selected item. The *Show HED schema* button brings up a browser for the HED vocabulary.

Through the CTagger GUI, users can explore the HED schema, quickly look up and add tags (or tag groups) to the desired event codes, and use import/export features to reuse tags on from other data recordings in the same study.

The process of tagging is simply choosing tags from the available vocabulary (using the HED schema browser) and associating these tags with each event code.

Once familiar with HED and the vocabulary, most users just type the tags directly in the tag window shown on the right.

CTagger is used as part of the HEDTools plug-in in this tutorial, but it can also be used as a standalone application.

Instructions on downloading and using the standalone version of CTagger, as well as step-by-step guide on how to add HED annotation with CTagger, can be found at in ***Tagging with CTagger***.

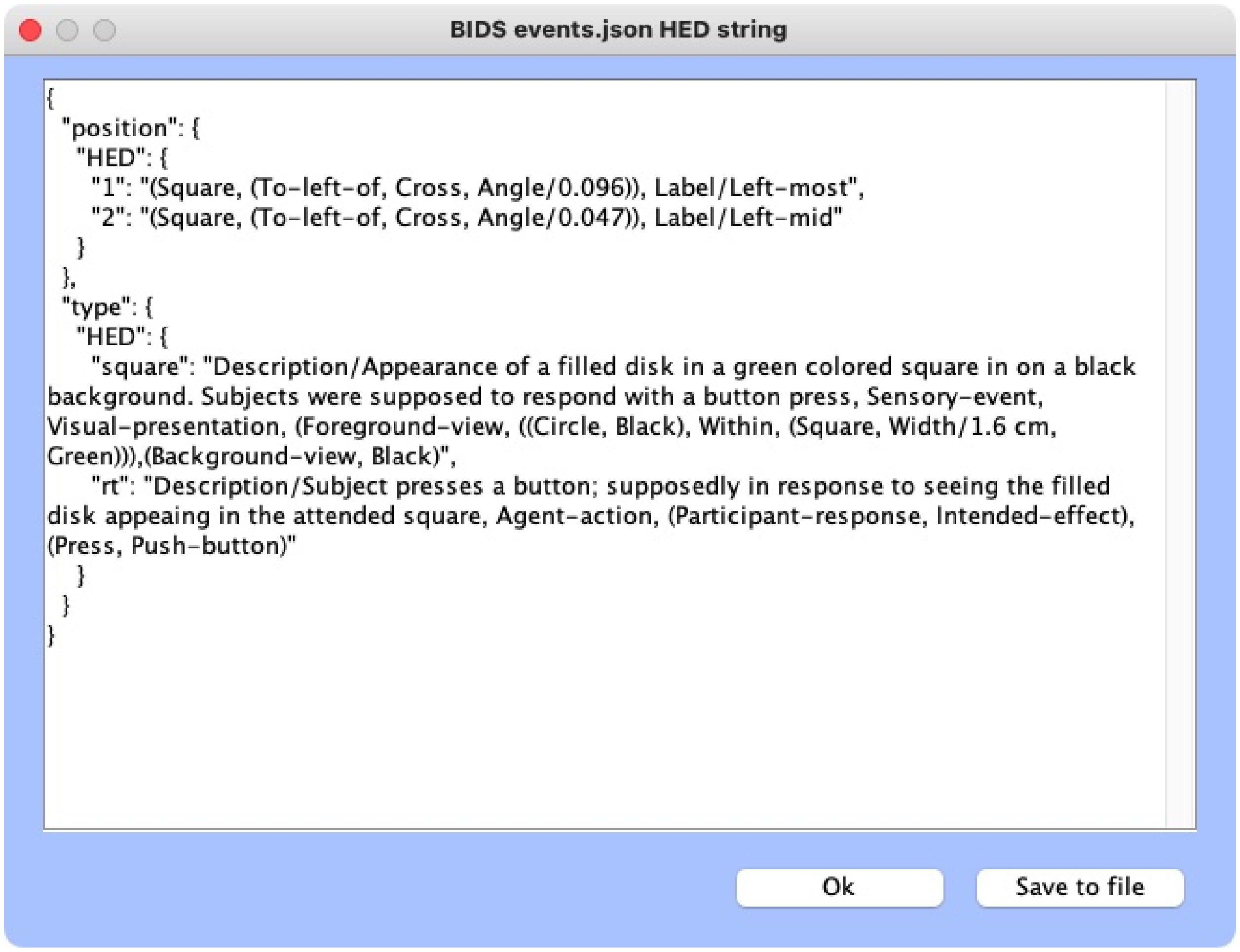
Tagging the events

A brief step-by-step guide to selecting tags can be found at ***HED annotation quickstart***. The following shows example annotations using the process suggested in the quickstart. we will import the annotation saved in the _events.json file format. Download the file **eeglab-tutorial_events.json** then select:

> **File –> Import –> Import BIDS events.json file**

to import it to CTagger. You can now review all the tags via:

> **File –> Review all tags**

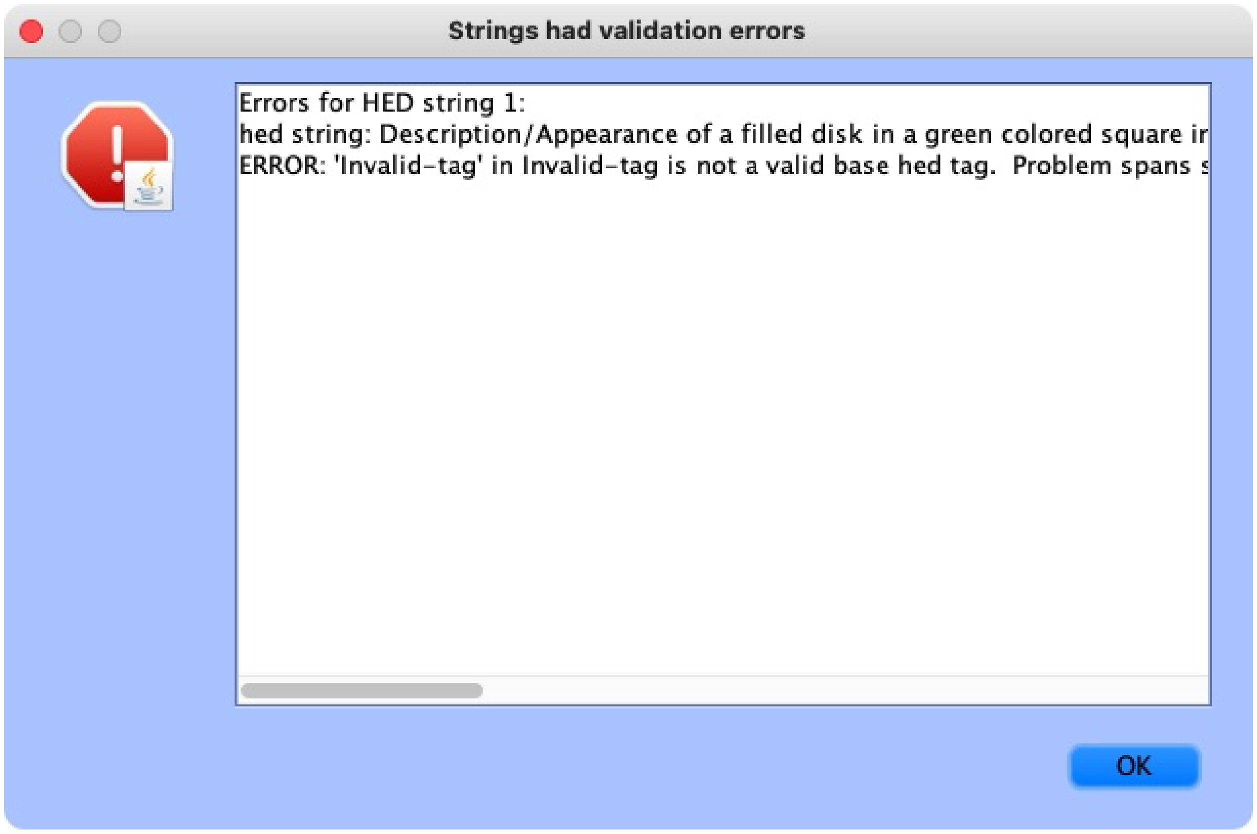


```
{
  "position": {
    "HED": {
      "1": "(Square, (To-left-of, Cross, Angle/0.096)), Label/Left-most",
      "2": "(Square, (To-left-of, Cross, Angle/0.047)), Label/Left-mid"
    }
  },
  "type": {
    "HED": {
      "square": "Description/Appearance of a filled disk in a green colored square in on a black
background. Subjects were supposed to respond with a button press, Sensory-event,
Visual-presentation, (Foreground-view, ((Circle, Black), Within, (Square, Width/1.6 cm,
Green))),(Background-view, Black)",
      "rt": "Description/Subject presses a button; supposedly in response to seeing the filled
disk appeaing in the attended square, Agent-action, (Participant-response, Intended-effect),
(Press, Push-button)"
    }
  }
}
```

Validation

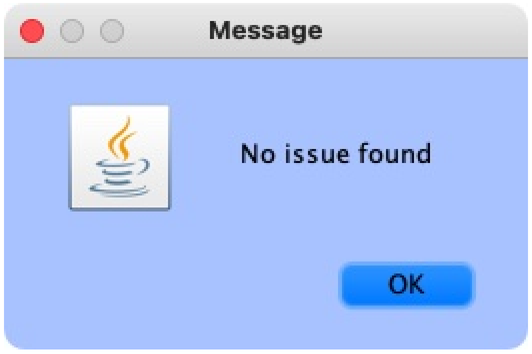
The last step of the annotation process is to validate the HED annotations. Click on the *Validate all* button at the bottom pane. A window will pop up showing validation results. If there are issues with the annotation, there will be a line for each of the issues found.

Here is an example of validation log file with issues:


Strings had validation errors

Errors for HED string 1:
hed string: Description/Appearance of a filled disk in a green colored square ir
ERROR: 'Invalid-tag' in Invalid-tag is not a valid base hed tag. Problem spans s

If the annotation was correct, a message will appear confirming the validity:

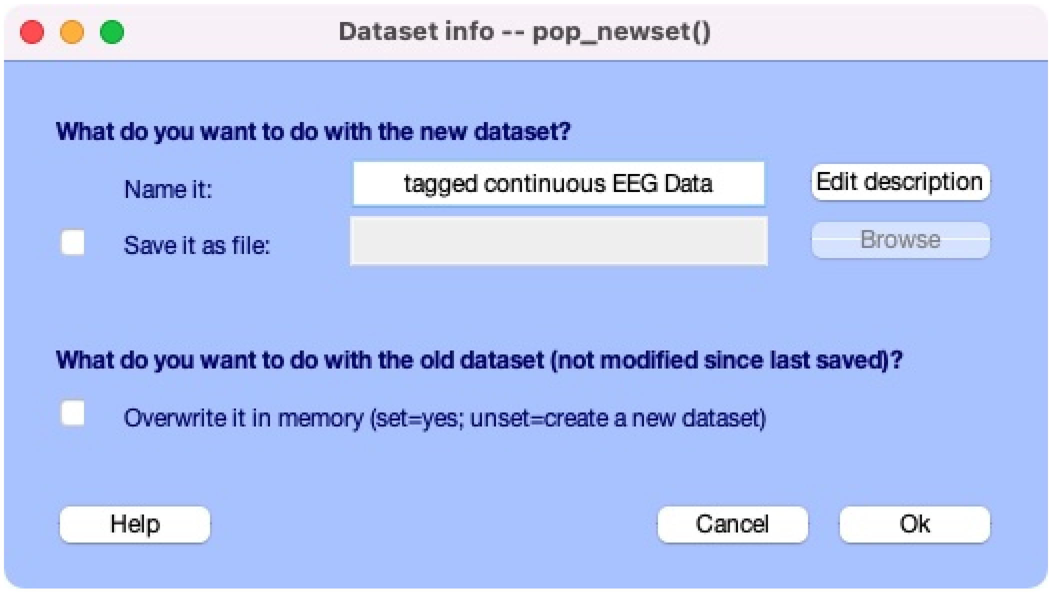

Message

No issue found

Click *Finish* on the main CTagger window to end the annotation.

The tag review window will show up again for a final review and the option to save the annotation into an _events.json file for distribution just as with the *eeglab-tutorial_events.json*. Hit *Ok* to continue after that.

A last window will pop up asking what you would like to overwrite the old dataset with the tagged one or save new dataset as a separate file. Click **Ok** when you're done.


Dataset info -- pop_newset()
What do you want to do with the new dataset?
Name it:
Save it as file:
Edit description
Browse
What do you want to do with the old dataset (not modified since last saved)?
Overwrite it in memory (set=yes; unset=create a new dataset)
Help
Cancel
Ok

Name it: tagged continuous EEG Data

You just finished tagging! *HEDTools* generates the final HED string for each event by concatenating all tags associated with the event values of that event (separated by commas). The final concatenated version is put the string in a new field **HED** in EEG.event.

6.17.2.3 HED-based epoching

The EEGLAB *pop_epoch* function extracts data epochs that are time locked to specified event types. This function allows you to epoch on one of a specified list of event types as defined by the *EEG.event.type* field of the EEG structure.

HEDTools provides a simple way for extracting data epochs from annotated datasets using a much richer set of conditions. To use HED epoching, you must have annotated the EEG dataset.

If the dataset is not tagged, please refer to *Annotating datasets* on how to tag a dataset.

Start by choosing the menu option: **Tools –> Extract epochs by tags**:

This will bring up a window to specify the options for extracting data epochs:

The *pop_epochhed* menu is almost identical to the EEGLAB *pop_epoch* menu with the exceptions of the first input field (**Time-locking HED tag(s)**) and the second input field (**Exclusive HED tag(s)**).

Instead of passing in or selecting from a group of unique event types, the user passes in a comma separated list of HED tags. For each event all HED tags in this list must be found for a data epoch to be generated.

Clicking the adjacent button (with the label . . .) will open a search tool to help you select HED tags retrieved from the dataset.

When you type something in the search bar, the dialog displays a list below containing possible matches. Pressing the "up" and "down" arrows on the keyboard while the cursor is in the search bar moves to the next or previous tag in the list.

Pressing "Enter" selects the current tag in the list and adds the tag to the search bar. You can continue search and add tags after adding a comma after each tag. When done, click the **Ok** button to return to the main epoching menu.

6.17.3 Python HEDTools in MATLAB

You can run functions from the Python `hedtools` library directly in MATLAB versions R2019a or later. With these tools you can incorporate validation, summary, search, factorization, and other HED processing directly into your MATLAB processing scripts without reimplementing these operations in MATLAB.

Note: For your reference, the source for `hedtools` is the **hed-python** GitHub repository. The code is fully open-source with an MIT license. The actual API documentation is available **here**, but the tutorials and tool documentation for `hedtools` on **HED Resources** provides more examples of use.

6.17.3.1 Getting started

The `hedtools` library requires a Python version >= 3.7. In order to call functions from this library in MATLAB, you must be running MATLAB version >= R2019a and have a **compatible version of Python** installed on your machine.

The most difficult part of the process for users who are unfamiliar with Python is getting Python connected to MATLAB. Once that is done, many of the standard `hedtools` functions have **MATLAB wrapper functions**, which take MATLAB variables as arguments and return MATLAB variables. Thus, once the setup is done, you don't have to learn any additional Python syntax to use the tools. You should only have to do this setup once, since MATLAB retains the setup information from session to session.

Steps for setting up Python HEDtools for MATLAB.

Step 1: Find Python. If yes, skip to Step 3.

Step 2: Install Python if needed .

Step 3: Connect Python to MATLAB. If already connected, skip to Step 4.

Step 4: Install HEDtools

Step 1: Find Python

Follow these steps until you find a Python executable that is version 3.7 or greater. If you can't locate one, you will need to install it.

Does MATLAB already have a good version of Python you can use?

In your MATLAB command window execute the following function:

```
pyenv
```

The following example response shows that MATLAB is using Python version 3.9 with executable located at `C:\Program Files\Python39\pythonw.exe`.

```
  PythonEnvironment with properties:

          Version: "3.9"
       Executable: "C:\Program Files\Python39\pythonw.exe"
          Library: "C:\Program Files\Python39\python39.dll"
             Home: "C:\Program Files\Python39"
           Status: NotLoaded
    ExecutionMode: InProcess
```

If MATLAB has already knows about a suitable Python version that is at least 3.7, you are ready to go to ***Step 4: Install HEDTools***. Keep track of the location of the Python executable.

If the `pyenv` did not indicate a suitable Python version, you will need to find the Python on your system (if there is one), or install your own.

There are several likely places to look for Python on your system.

For Linux users:

> Likely places for system-space installation are `/bin`, `/local/bin`, `/usr/bin`, `/usr/local/bin`, or `/opt/bin`. User-space installations are usually your home directory in a subdirectory such as `~/bin` or `~/.local/bin`.

For Windows users:

> Likely places for system-space installation are `C:\`, `C:\Python`, or `C:\Program Files`. User-space installations default to your personal account in `C:\Users\yourname\AppData\Local\Programs\Python\python39` where `yourname` is your Windows account name and `python39` will be the particular version (in this case Python 3.9).

If you don't have any success finding a Python executable, you will need to install Python as described in ***Step 2: Install Python if needed***.

Otherwise, you can skip to ***Step 3:Connect Python to MATLAB***.

Warning: You need to keep track of the path to your Python executable for Step 3.

Step 2: Install Python if needed

If you don't have Python on your system, you will need to install it. Go to **Python downloads** and pick the correct installer for your operating system and version.

Depending on your OS and the installer options you selected, Python may be installed in your user space or in system space for all users.

- You should keep track of the directory that Python was installed in.
- You may want to add the location of the Python executable to your PATH. (Most installers give you that option as part of the installation.)

Step 3: Connect Python to Matlab

Setting the Python version uses the MATLAB `pyenv` function with the `'Version'` argument as illustrated by the following example.

Example MATLAB function call connect MATLAB to Python.

```
>> pyenv('Version', 'C:\Program Files\Python39\python.exe')
```

Be sure to substitute the path of the Python that you have found. Notice that the executable listed in Step 1 was `pythonw.exe`, but we have used `python.exe` here to indicate the command line version.

Use the MATLAB `pyenv` function again without arguments to check that your installation is as expected.

Example response for pyenv all with no argments after setting environment.

```
PythonEnvironment with properties:

          Version: "3.9"
       Executable: "C:\Program Files\Python39\python.exe"
          Library: "C:\Program Files\Python39\python39.dll"
             Home: "C:\Program Files\Python39"
           Status: NotLoaded
    ExecutionMode: InProcess
```

Step 4: Install HEDTools

The general-purpose package manager for Python is called `pip`. By default, `pip` retrieves packages to be installed from the **PyPI** package repository. You will need to use the version of `pip` that corresponds to the version of Python that is connected to MATLAB. This may not be the default `pip` used from the command line.

Command to install hedtools in MATLAB.

To install the latest released version of `hedtools` type a pip command such as the following in your MATLAB command window.

```
system('"C:\Program Files\Python39\Scripts\pip" install hedtools')
```

Use the full path of the pip associated with the Python that you are using with MATLAB

Giving the full path to `pip` corresponding to the Python installation that MATLAB is using ensures that MATLAB knows about `HEDtools`. (The version of MATLAB that Python is using may not be the same as the Python in the system PATH.)

Also watch the resulting messages in the command window to make sure that HEDtools was successfully installed. In the case of the above example, the Python being used is in system space, which requires administrator privileges.

The first line of the output was:

```
   Defaulting to user installation because normal site-packages is not writeable
```

On Windows these packages will be found in a `site-packages` directory such as:

```
`C:\Users\username\AppData\Roaming\Python\Python39\site-packages`
```

On Linux these packages might be found in directory such as:

```
/home/username/.local/lib/python3.9/site-packages/
```

Warning: If your system had a Python 2 installed at some point, your Python 3 executable might be named `python3` rather than `python`.

Similarly, the `pip` package manager might be named `pip3` instead of `pip`.

The following MATLAB statement can be used to test that everything was installed correctly.

Test that everything is installed.

```
pyrun("from hed import _version as vr; print(f'Using HEDTOOLS version: {str(vr.get_
↪versions())}')")
```

If everything installed correctly, the output will be something like

```
Using HEDTOOLS version: {'date': '2022-06-20T14:40:24-0500', 'dirty': False, 'error':␣
↪None, 'full-revisionid': 'c4ecd1834cd31a05ebad3e97dc57e537550da044', 'version': '0.1.0
↪'}
```

6.17.3.2 MATLAB wrappers for HEDTools

The **hedtools_wrappers** directory in the **hed-examples** GitHub repository contains MATLAB wrapper functions for calling various commonly used HED tools.

Direct calls to HEDTools

Wrapper functions are provided to some of the more commonly used functions in the HEDTools suite.

The following example shows the MATLAB wrapper function **validateHedInBids.m**, which contains the underlying calls to HEDTools Python BIDs validation.

A MATLAB wrapper function for validating HED in a BIDS dataset.

```
function issueString = validateHedInBids(dataPath)
    py.importlib.import_module('hed');
    bids = py.hed.tools.BidsDataset(dataPath);
    issues = bids.validate();
    issueString = string(py.hed.get_printable_issue_string(issues));
```

Example MATLAB calling code for this function:

```
dataPath = 'H:\datasets\eeg_ds003645s_hed';
issueString = validateHedInBids(dataPath);
if isempty(issueString)
    fprintf('Dataset %s has no HED validation errors\n', dataPath);
else
    fprintf('Validation errors for dataset %s:\n%s\n', dataPath, issueString);
end
```

In above example assumes that the BIDS dataset was located at `H:\datasets\eeg_ds003645s_hed`. We tested it with the **eeg_ds003645s_hed** available on GitHub. You can download and use this test data or set `dataPath` to the root directory of your own dataset.

Calls to HED remodeling tools

Many of the most useful HEDTools functions are packaged in the HED remodeling tool suite. These tools allow operations such as creating summaries, validating the dataset, and transforming event files to be run on an entire dataset.

The following example illustrates a call that creates a summary of the experimental conditions for a HED-tagged dataset.

A MATLAB wrapper function for a remodeling operation to create a summary.

```
function runRemodel(remodel_args)
    py.importlib.import_module('hed');
    py.hed.tools.remodeling.cli.run_remodel.main(remodel_args);
```

Example MATLAB calling code for this function:

```
dataPath = 'G:\ds003645';
remodelFile = 'G:\summarize_hed_types_rmdl.json';
remodel_args = {dataPath, remodelFile, '-b', '-x', 'stimuli', 'derivatives'};
runRemodel(remodel_args);
```

The command line arguments to the various remodeling functions are given in a cell array, rather than a regular MATLAB array. For the remodeling operations, first and second operation must be the dataset root directory and the remodeling file name, respectively. In this example, dataset `ds003645` has been downloaded from **openNeuro** to the `G:\` drive. The remodeling file used in this example can be found at See ***File remodeling quickstart*** and ***File remodeling tools*** for additional information. The wrapper functions are available on GitHub in the **hedtools_wrappers** directory.

6.17.3.3 MATLAB functions for Python

The following table lists the relevant MATLAB functions that are available. You should refer to the help facility for your version of MATLAB to get the details of what is supported for your version of MATLAB.

MATLAB command	Purpose
`pyenv`	Setup your Python environment in MATLAB.Without arguments outputs information about your current Python environment.
`pyrun`	Run a Python statement and return results.
`pyargs`	A recent addition for more advanced argument handling.
`pyrunfile`	Run a Python script from MATLAB.

The MATLAB `matlab.exception.PyException` captures error information generated during Python execution.

6.18 HED schemas

6.18.1 HED schema basics

HED annotations consist of unordered comma separated lists of HED tags. The annotations may include parentheses to group terms that belong together. For example in the HED annotation *Red, Triangle, Blue, Square*, cannot use ordering to determine which tags belong together. To indicate a red triangle and a blue square, you must use parentheses: *(Red, Triangle), (Blue, Square)*.

The HED tags used to annotate data come from a controlled vocabulary called a **HED schema**. A HED schema consists of a series of **top-level** tags representing general categories in this vocabulary. Each top-level tag is the root of a tree containing tags falling into that category.

Each child tag in a HED schema is considered to be a special type of its ancestors. Consider the tag *Square*, which has a full schema path *Item/Object/Geometric-object/2D-shape/Rectangle/Square*. *Square* is-a type of *Rectangle*, which is-a type of *2D-shape*, etc.

The strict requirement of child **is-a** type of any ancestor means that when downstream tools search for *2D-shape*, the search will return tag strings containing *Square* as well as those containing the tags *Rectangle* and *2D-shape*.

Rules for the HED schema vocabulary and for HED-compliant tools can be found in the **HED Specification**.

Additional details about particular schemas can be found in the **HED schemas** documentation page.

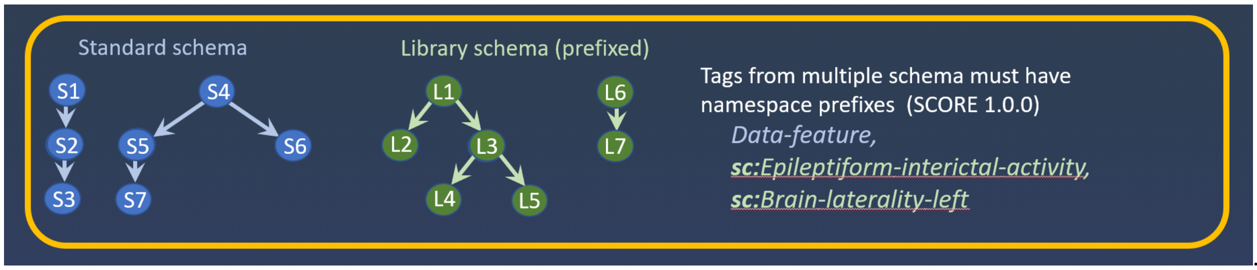
6.18.1.1 Tag forms

When you tag, you need only use the tag node name (e.g, *Square*). **HED-compliant tools** can convert between this "short-form" and the complete path or "long-form" (e.g. */Item/Object/Geometric-object/2D-shape/Rectangle/Square*) when needed for search, summarization, or other processing.

6.18.1.2 Types of schemas

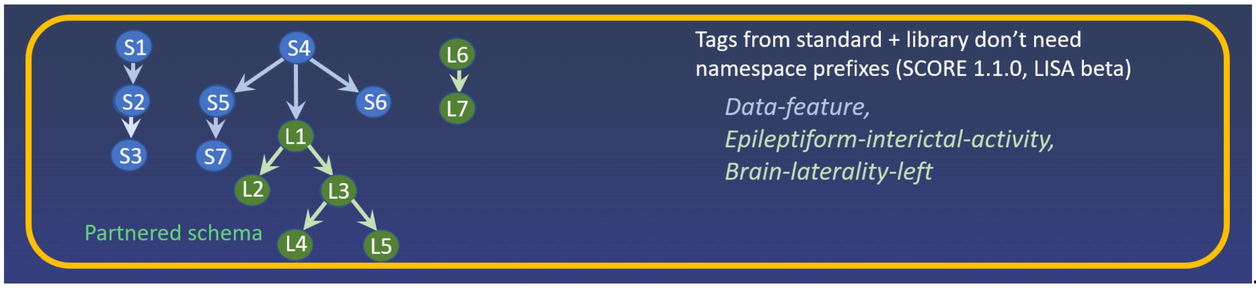
The HED standard schema consists of a terms that are likely to be of use in all experiments, while library schemas capture the terms that are important for annotations in a specialized areas. You may use terms from as many schemas as you wish. However, if you use more than one schema, terms from the additional schemas must be prefixed by **local namespace designators** to indicate which schemas the tags came from. A namespace designator is the form *xx:* where *xx* is a user-chosen string of alphabetic characters.

The following diagram shows a representation of a standard schema (blue nodes) used in conjunction with the SCORE 1.0.0 library schema (green nodes). Tags from the standard schema, such as *Data-feature* appear without the prefix. The remaining tags, which come from the SCORE library, appear with the user-defined *sc:* namespace prefix in the annotation.


Standard schema
S1
S2
S3
S4
S5
S6
S7
Library schema (prefixed)
L1
L2
L3
L4
L5
L6
L7
Tags from multiple schema must have namespace prefixes (SCORE 1.0.0)
Data-feature,
sc:Epileptiform-interictal-activity,
sc:Brain-laterality-left

.

Starting with HED schema version **8.2.0**, HED supports **partnered schemas**, which are library schemas that are merged with a standard schema. Partnered schemas allow schema designers to include library tags that are elaborations of tags in the standard schema in addition to other specialized tags as shown in the following diagram:


S1
S2
S3
S4
S5
S6
S7
L1
L2
L3
L4
L5
L6
L7
Partnered schema
Tags from standard + library don't need namespace prefixes (SCORE 1.1.0, LISA beta)
Data-feature,
Epileptiform-interictal-activity,
Brain-laterality-left

.

SCORE version 1.1.0 will be distributed as a partnered schema. Annotations from a partnered schema can include tags from both the library schema and its partner without prefixes.

6.18.2 Viewing schemas

All versions of the HED schemas are located in the GitHub **hed-schemas** and can be best-viewed using the **HED schema browser**.

6.18.3 Available schemas

6.18.3.1 The standard schema

The HED standard schema contains the basic vocabulary for annotating experiments. These are terms that are likely to be useful in all types of annotations. The HED standard schema source is located in the **standard_schema** directory of the **hed-schemas** GitHub repository.

Format	Type	Use
XML	**Raw**	Accessed by tools for validation and analysis.
	Formatted	Readable display.
Mediawiki	**Raw**	Edited to create a new schema.
	Formatted	Readable display for editing.
Prerelease	**Directory**	Working directory for developing the prerelease.

6.18.3.2 The SCORE library

The HED SCORE library is an implementation of the **SCORE** standard for clinical annotation of EEG by neurologists. For more information and the latest references see **HED SCORE schema**.

Format	Type	Use
XML	**Raw**	Accessed by tools for validation and analysis.
	Formatted	Readable display.
Mediawiki	**Raw**	Edited to create a new schema.
	Formatted	Readable display for editing.
Prerelease	**Directory**	Working directory for developing the prerelease.

6.18.3.3 The LISA library

The HED LISA library represents a vocabulary for annotating linguistic stimuli in language and other types of experiments. For more information and the latest references see **HED LISA schema**. The LISA library is under development and is only available in prerelease format.

Format	Type	Use
XML	**Raw**	Accessed by tools for validation and analysis.
	[**Formatted**]	Readable display.
Mediawiki	[**Raw**]	Edited to create a new schema.
	[**Formatted**]	Readable display for editing.
Prerelease	**Directory**	Working directory for developing the prerelease.

6.19 HED test datasets

The **hed-examples** repository contains a set of HED-annotated datasets in **BIDS**-compatible format. These datasets can be useful for:

1. Writing lightweight software tests.
2. Serving as examples of how to incorporate HED into BIDS-structured data.

The datasets have **empty raw data files**. However, some data headers containing the metadata are still intact.

Datasets that are derived from datasets on OpenNeuro are identified by their OpenNeuro accession number plus 's' plus a modifier. Datasets focused on particular a particular modality may have the modality prepended to the name. For example, eeg_ds003645s identifies a reduced dataset derived the EEG data in OpenNeuro dataset ds003645. The suffix modifier indicates what this dataset is designed to test.

Dataset	Description	OpenNeuro
eeg_ds002893s_hed_attention_shift	Shift between auditory and visual modalities.	**ds002893**
eeg_ds003645s_hed	Short-form tags with definitions.	**ds003645**
eeg_ds003645s_hed_column	Some events.tsv files contain a `HED` column.	
eeg_ds003645s_hed_inheritance	Multiple sidecars with inheritance.	
eeg_ds003645s_hed_library	Multiple HED library schemas.	
eeg_ds003645s_hed_longform	Long-form with definitions.	
eeg_ds004105s_hed_longform	BCIT auditory cueing	**ds004105**
eeg_ds004106s_hed_longform	BCIT advanced guard duty	**ds004106**
eeg_ds004117s_hed_sternberg	Sternberg working memory task	**ds004117**
fmri_ds002790s_hed_aomic	Annotation with single column.	**ds002790**
fmri_soccer21_hed	Annotation with single column.	

6.19.1 eeg_ds002893s_hed

This dataset includes rapid shifts in instructed attention between visual and auditory modalities. The dataset is mentioned as an example in the OHBM 2022 tutorial **Annotating the timeline of neuroimaging time series data using Hierarchical Event Descriptors**.

6.19.2 eeg_ds003645s_hed

This dataset was originally released as multi-modal dataset **ds000117** by Daniel Wakeman and Richard Henson. The dataset events in **ds003645** have been reorganized from the original and additional events added from the experimental logs. The dataset includes MEEG and behavioral data. HED tags have been added.

The dataset is used as a HED case study in:

> Robbins, K., Truong, D., Appelhoff, S., Delorme, A., & Makeig, S. (2021).
> Capturing the nature of events and event context using Hierarchical Event Descriptors (HED).
> Neuroimage 2021 Dec 15;245:118766. doi: 10.1016/j.neuroimage.2021.118766. Epub 2021 Nov 27.
> https://www.sciencedirect.com/science/article/pii/S1053811921010387?via%3Dihub.

6.19.3 eeg_ds003645s_hed_column

This is a modification of ds003645s_hed where some `events.tsv` files contain a `HED` column.

6.19.4 eeg_ds003645s_hed_inheritance

This is a modification of ds003645s_hed where multiple sidecars containing HED tags are included to test that HED tools correctly handle BIDS inheritance rules.

6.19.5 eeg_ds003645s_hed_library

This dataset is designed to test the HED library schema facility. It uses HED 8.0.0 as a base schema and as the "test" library schema. In addition, this dataset uses the SCORE library version 1.0.0 as a library schema.

The schemas are specified in the `dataset_description.json` file.

6.19.6 eeg_ds003645s_hed_longform

This is a modification of ds003645s_hed where the HED tags include a mix of tags in long and short forms to test that tools work with either long-form or short-form HED tags.

6.19.7 eeg_ds004105s_hed

Subjects in the Auditory Cueing study performed a long-duration simulated driving task with perturbations and audio stimuli in a visually sparse environment. The dataset is part of a collection of 10 datasets from the BCIT program designed to test EEG mega-analysis.

6.19.8 eeg_ds004106s_hed

BCIT Advanced Guard Duty study was designed to measure sustained vigilance in realistic settings by having subjects verify information on replica ID badges. The dataset is part of a collection of 10 datasets from the BCIT program designed to test EEG mega-analysis.

6.19.9 eeg_ds004117s_hed_sternberg

Sternberg working memory dataset, described in **Onton et al. 2005**, is used in a number of HED case studies including the OHBM 2022 tutorial **Annotating the timeline of neuroimaging time series data using Hierarchical Event Descriptors** and the book chapter **2.3 End-to-end processing of M/EEG data with BIDS, HED, and EEGLAB** by Thruong et al. in **Methods for analyzing large neuroimaging datasets** edited by Whelan and Lemaitre.

The study was also selected for replication in the **EEGManyLabs** initiative.

6.19.10 fmri_ds002790s_hed_aomic

This dataset is part of the **Amsterdam OpenMRI Collection (AOMIC)**.

The dataset is used as a case study for the book chapter **2.4 Actionable event annotation and analysis in fMRI: A practical guide to event handling** by Denissen et al. in **Methods for analyzing large neuroimaging datasets** edited by Whelan and Lemaitre.

6.19.11 fmri_soccer21_hed

This dataset is designed to illustrate a basic FMRI pipeline. The dataset is used as a case study for the book chapter **2.4 Actionable event annotation and analysis in fMRI: A practical guide to event handling** by Denissen et al. in **Methods for analyzing large neuroimaging datasets** edited by Whelan and Lemaitre.

6.19.12 BIDS validation

For general information on the `bids-validator`, including installation, configuration, and usage, see the bids-validator README file.

Example: The following command validates the `eeg_ds003645s_hed` dataset:

```
bids-validator eeg_ds003645s_hed --config.ignore=99
```

This example assumes that `npm` and the `bids-validator` npm package have been installed on the local machine. The command is run from the directory above the dataset root directory. The `--config.ignore=99` flag tells the bids-validator to ignore empty data files rather than to report the empty file error.

For additional information on BIDS validation, see the bids-examples.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search