Snakebids

Release 0.13.0

Ali R. Khan

Apr 30, 2024

USER GUIDE

Warning: Snakebids is migrating to a more robust, extensible API! If you're coming from pre-v0.8 code, check out the *migration guide* to ensure your workflow is up-to-date.

Snakebids is a Python package that extends Snakemake, enabling users to create reproducible, scalable pipelines for processing neuroimaging data in the BIDS format. Snakebids workflows expose a CLI that conforms to the BIDS App guidelines.

CHAPTER

ONE

FEATURES

Snakebids includes all of the features of Snakemake, including flexible configuration, parallel execution, and Docker/Singularity support, plus:

- **Built-in support for BIDS datasets**: Seamless workflow functionality with a wide range of BIDS datasets, accomodating various levels of complexity.
- **BIDS App Creation**: Provide command-line invocations of your workflow following BIDS App guidelines, ensuring reproducibility and enhancing accessibility of your workflow.
- **BIDS Path Construction**: Easy, flexible construction of valid BIDS paths following BIDS guiding principles, promoting data organization and sharing.
- **Plugin System**: Extend the functionality of Snakebids by creating and using plugins to meet your workflow's needs.
- **Pybids Querying**: Leverages Pybids to efficiently retrieve specific data required.

CHAPTER

TWO

INSTALLATION

Snakebids can be installed using pip:

```
pip install snakebids
```

CHAPTER

THREE

USAGE

To create and run a Snakebids workflow, you need to:

1. **Create a Snakefile**: Define the steps of your workflow, including input / output files, processing rules, and dependencies
2. **Create a configuration file**: Customize workflow behaviour using a YAML configuration file. Specify input / output directories and custom workflow parameters.
3. **Run the pipeline**: Execute the Snakebids pipeline by invoking the BIDS App CLI or via Snakemake executable.

For detailed instructions and examples, please refer to the **documentation**.

CHAPTER

FOUR

CONTRIBUTING

Snakebids is an open-source project, and contributions are welcome! If you have any bug reports, feature requests, or improvements, please submit them to the **issues page**.

To contribute, first clone the Github repository. Snakebids dependencies are managed with Poetry (version 1.2 or higher). Please refer to the poetry website for installation instructions.

Note: Snakebids makes use of Poetry's dynamic versioning. To see a version number on locally installed Snakebids versions, you will have to also install `poetry-dynamic-versioning` *plugin to your poetry installation (`poetry self add "poetry-dynamic-versioning[plugin]"). This is* ***not required*** *for contribution.*

Following installation of Poetry, the development can be set up by running the following commands:

```
poetry install
poetry run poe setup
```

Snakebids uses poethepoet as a task runner. You can see what commands are available by running:

```
poetry run poe
```

Tests are done with `pytest` and can be run via:

```
poetry run poe test
```

Additionally, Snakebids uses pre-commit hooks (installed via the `poe setup` command above) to lint and format code (we use black, isort and ruff. By default, these hooks are run on every commit. Please be sure they all pass before making a PR.

CHAPTER

FIVE

LICENSE

Snakebids is distributed under the MIT License.

CHAPTER

SIX

ACKNOWLEDGEMENTS

Snakebids extends the Snakemake workflow management system and follows the guidelines outlined by the BIDS specification.

CHAPTER

SEVEN

RELEVANT PAPERS

- Mölder F, Jablonski KP, Letcher B et al. Sustainable data analysis with Snakemake [version 2; peer review: 2 approved]. F1000Research. 2021. doi: 10.12688/f1000research.29032.2

7.1 Why use snakebids?

Snakebids makes it easy to use Snakemake to break down a neuroimaging workflow into its component steps, while still providing a the standard command line interface of a BIDS app.

7.2 Tutorial

7.2.1 Getting started

In this example we will make a workflow to smooth `bold` scans from a bids dataset.

We will start by creating a simple rule, then make this more generalizable in each step. To begin with, this is the command we are using to smooth a bold scan.

```
fslmaths ../bids/sub-001/func/sub-001_task-rest_run-1_bold.nii.gz -s 2.12 results/sub-
↪001/func/sub-001_task-rest_run-1_fwhm-5mm_bold.nii.gz
```

This command performs smoothing with a sigma=2.12 Gaussian kernel (equivalent to 5mm FWHM, with sigma=fwhm/2.355), and saves the smoothed file as `results/sub-001/func/sub-001_task-rest_run-1_fwhm-5mm_bold.nii.gz`.

Installation

Start by making a new directory:

```
$ mkdir snakebids-tutorial
$ cd snakebids-tutorial
```

Check your python version to make sure you have at least version 3.7 or higher:

```
$ python --version
Python 3.10.0
```

Make a new virtual environment:

```
$ python -m venv .venv
$ source .venv/bin/activate
```

And use pip to install snakebids:

```
$ pip install snakebids
```

In our example, we'll be using the `fslmaths` tool from *FSL*. If you want to actually run the workflow, you'll need to have FSL installed. This is not actually necessary to follow along the tutorial however, as we can use "dry runs" to see what snakemake *would* do if FSL were installed.

Getting the dataset

We will be running the tutorial on a test dataset consisting only of empty files. We won't actually be able to run our workflow on it (`fslmaths` will fail), but as mentioned above, we can use dry runs to see would would normally happen.

If you wish to follow along using the same dataset, currently the easiest way is to start by cloning snakebids:

```
$ git clone https://github.com/khanlab/snakebids.git
```

Then copy the following directory:

```
$ cp -r snakebids/docs/tutorial/bids ./data
```

It's also perfectly possible (and probably better!) to try the tutorial on your own dataset. Just adjust any paths below so that they match your data!

7.2.2 Part I: Snakemake

Step 0: a basic non-generic workflow

In this rule, we start by creating a rule that is effectively hard-coding the paths for input and output to re-create the command as above.

In this rule we have an `input:` section for input **files**, a `params:` section for **non-file** parameters, and an `output:` section for output files. The shell section is used to build the shell command, and can refer to the input, output or params using curly braces. Note that any of these can also be named inputs, but we have only used this for the `sigma` parameter in this case.

Listing 1: Snakefile

```
rule smooth:
    input:
        'data/sub-001/func/sub-001_task-rest_run-1_bold.nii.gz'
    params:
        sigma = '2.12'
    output:
        'results/sub-001/func/sub-001_task-rest_run-1_fwhm-5mm_bold.nii.gz'
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

With this rule in our Snakefile, we can then run `snakemake -np` to execute a dry-run of the workflow. Here the `-n` specifies dry-run, and the `-p` prints any shell commands that are to be executed.

When we invoke `snakemake`, it uses the first rule in the snakefile as the `target` rule. The target rule is what snakemake uses as a starting point to determine what other rules need to be run in order to generate the inputs. We'll learn a bit more about that in the next step.

So far, we just have a fancy way of specifying the exact same command we started with, so there is no added benefit (yet). But we will soon add to this rule to make it more generalizable.

Step 1: adding wildcards

First step to make the workflow generalizeable is to replace the hard-coded identifiers (e.g. the subject, task and run) with wildcards.

In the Snakefile, we can replace `sub-001` with `sub-{subject}`, and so forth for task and run. Now the rule is generic for any subject, task, or run.

Listing 2: Snakefile

```
rule smooth:
    input:
        'data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.nii.gz'
    params:
        sigma = '2.12'
    output:
        'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-5mm_bold.
↪nii.gz'
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

However, if we try to execute (dry-run) the workflow as before, we get an error. This is because the `target` rule now has wildcards in it. So snakemake is unable to determine what rules need to be run to generate the inputs, since the wildcards can take any value.

So for the time being, we will make use of the snakemake command-line argument to specify `targets`, and specify the file we want generated from the command-line, by running:

```
$ snakemake -np results/sub-001/func/sub-001_task-rest_run-1_fwhm-5mm_bold.nii.gz
```

We can now even try running this for another subject by changing the target file.

```
$ snakemake -np results/sub-002/func/sub-002_task-rest_run-1_fwhm-5mm_bold.nii.gz
```

Try using a subject that doesn't exist in our bids dataset, what happens?

Now, try changing the output smoothing value, e.g. `fwhm-10mm`, and see what happens. As expected the command still uses a smoothing value of 2.12, since that has been hard-coded, but we will see how to rectify this in the next step.

Step 2: adding a params function

As we noted, the sigma parameter needs to be computed from the FWHM. We can use a function to do this. Functions can be used for any `input` or `params`, and must take `wildcards` as an input argument, which provides a mechanism to pass the wildcards (determined from the output file) to the function.

We can thus define a simple function that returns a string representing `FWHM/2.355` as follows:

Listing 3: Snakefile

```
def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'
```

Note 1: We now have to make the fwhm in the output filename a wildcard, so that it can be passed to the function (via the wildcards object).

Note 2: We have to convert the fwhm to float, since all wildcards are always strings (since they are parsed from the output filename).

Once we have this function, we can replace the hardcoded `2.12` with the name of the function:

Listing 4: Snakefile

```
        'data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.nii.gz'
    params:
        sigma = calc_sigma_from_fwhm
    output:
```

Here is the full Snakefile:

Listing 5: Snakefile

```
def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'

rule smooth:
    input:
        'data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.nii.gz'
    params:
        sigma = calc_sigma_from_fwhm
    output:
        'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}mm_
↪bold.nii.gz'
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

Now try running the workflow again, with `fwhm-5` as well as `fwhm-10`.

Step 3: adding a target rule

Now we have a generic rule, but it is pretty tedious to have to type out the filename of each target from the command-line in order to use it.

This is where target rules come in. If you recall from earlier, the first rule in a workflow is interpreted as the target rule, so we just need to add a dummy rule to the Snakefile that has all the target files as inputs. It is a dummy rule since it doesn't have any outputs or any command to run itself, but snakemake will take these input files, and determine if any other rules in the workflow can generate them (considering any wildcards too).

In this case, we have a BIDS dataset with two runs (run-1, run-2), and suppose we wanted to compute smoothing with several different FWHM kernels (5,10,15,20). We can thus make a target rule that has all these resulting filenames as inputs.

A very useful function in snakemake is `expand()`. It is a way to perform array expansion to create lists of strings (input filenames).

Listing 6: Snakefile

```
rule all:
    input:
        expand(
            'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}
↪mm_bold.nii.gz',
            subject='001',
            task='rest',
            run=[1,2],
            fwhm=[5,10,15,20]
        )


```

Now, we don't need to specify any targets from the command-line, and can just run:

```
$ snakemake -np
```

The entire Snakefile for reference is:

Listing 7: Snakefile

```
rule all:
    input:
        expand(
            'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}
↪mm_bold.nii.gz',
            subject='001',
            task='rest',
            run=[1,2],
            fwhm=[5,10,15,20]
        )


def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'

rule smooth:
```

(continues on next page)

(continued from previous page)

```
    input:
        'data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.nii.gz'
    params:
        sigma = calc_sigma_from_fwhm,
    output:
        'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}mm_
↪bold.nii.gz'
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

Step 4: adding a config file

We have a functional workflow, but suppose you need to configure or run it on another bids dataset with different subjects, tasks, runs, or you want to run it for different smoothing values. You have to actually modify your workflow in order to do this.

It is a better practice instead to keep your configuration variables separate from the actual workflow. Snakemake supports this by allowing for a separate config file (can be YAML or JSON, here we will use YAML), where we can store any dataset specific configuration. Then to apply it for a new purpose, you can simply update the config file.

To do this, we simply add a line to our workflow:

Listing 8: Snakefile

```
configfile: 'config.yml'

rule all:
    input:
```

Snakemake will then handle reading it in, and making the configuration variables available via dictionary called `config`.

In our config file, we will add variables for everything in the target rule `expand()`:

Listing 9: config.yaml

```
subjects:
  - '001'

tasks:
  - rest

runs:
  - 1
  - 2

fwhm:
  - 5
  - 10
  - 15
  - 20

```

(continues on next page)

(continued from previous page)

```

in_bold: 'data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.nii.gz'
```

In our Snakefile, we then need to replace these hardcoded values with `config[key]`. The entire updated Snakefile is shown here:

Listing 10: Snakefile

```
configfile: 'config.yml'

rule all:
    input:
        expand(
            'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}
↪mm_bold.nii.gz',
            subject=config['subjects'],
            task=config['tasks'],
            run=config['runs'],
            fwhm=config['fwhm']
        )


def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'

rule smooth:
    input:
        config['in_bold']
    params:
        sigma = calc_sigma_from_fwhm
    output:
        'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}mm_
↪bold.nii.gz'
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

After these changes, the workflow should still run just like the last step, but now you can make any changes via the config file.

7.2.3 Part II: Snakebids

Now that we have a fully functioning and generic Snakemake workflow, let's see what Snakebids can add.

Step 5: the bids() function

The first thing we can make use of is the *bids()* function. This provides an easy way to generate bids filenames. This is especially useful when defining output files in your workflow and you have many bids entities.

In our existing workflow, this was our output file:

Listing 11: Snakefile

```
    output:
        'results/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_fwhm-{fwhm}mm_
↪bold.nii.gz'
```

To create the same path using *bids()*, we just need to specify the root directory (`results`), all the bids tags (subject, task, run, fwhm), and the suffix (which includes the extension):

Listing 12: Snakefile

```
    output:
        bids(
            root='results',
            subject='{subject}',
            task='{task}',
            run='{run}',
            fwhm='{fwhm}',
            suffix='bold.nii.gz',
        )
```

Note: To make a snakemake wildcard, we wrapped the `'value'` in curly braces (e.g. `'{value}'`).

Note: The entities you supply in the *bids()* function do not have to be in the BIDS specification, e.g. `fwhm` is not in the spec. But if you do use entities that are in the BIDS specification, snakebids will ensure they go in the correct order (e.g. `sub_*-ses_*-run_*...`). Non-standard entities will be added to the end of the path, just before the suffix, in the order they were defined.

The Snakefile with the output filename replaced (in both rules) is below:

Listing 13: Snakefile

```
from snakebids import bids

configfile: 'config.yml'

rule all:
    input:
        expand(
            bids(
                root='results',
                subject='{subject}',
                task='{task}',
                run='{run}',
                fwhm='{fwhm}',
```

(continues on next page)

(continued from previous page)

```
                suffix='bold.nii.gz'
            ),
            subject=config['subjects'],
            task=config['tasks'],
            run=config['runs'],
            fwhm=config['fwhm'],
        )


def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'

rule smooth:
    input:
        config['in_bold']
    params:
        sigma = calc_sigma_from_fwhm
    output:
        bids(
            root='results',
            subject='{subject}',
            task='{task}',
            run='{run}',
            fwhm='{fwhm}',
            suffix='bold.nii.gz',
        )
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

Step 6: parsing the BIDS dataset

So far, we have had to manually enter the path to input bold file in the config file, and also specify what subjects, tasks, and runs we want processed. Can't we use the fact that we have a BIDS dataset to automate this a bit more?

With Snakemake, there are ways to glob the files to figure out what wildcards are present (e.g. `glob_wildcards()`), however, this is not so straightforward with BIDS, since filenames in BIDS often have optional components. E.g. some datasets may have a `ses` tag/sub-directory, and others do not. Also there are often optional user-defined values, such as the `acq` tag, that a workflow in most cases should ignore. Thus, the input that we use in our workflow, `in_bold`, that has wildcards to be generic, would need to be altered for any given BIDS dataset, along with the workflow itself, making this automated BIDS parsing difficult within Snakemake.

Snakebids lets you parse a bids dataset (using pybids under the hood) using a configfile that contains the required wildcards, along with data structures that specify all the wildcard values for all the subjects. This, in combination with the *`bids()`* function, can allow one to make snakemake workflows that are compatible with any general bids dataset.

To add this parsing to the workflow, we call the *`generate_inputs()`* function before our rules are defined, and pass along some configuration data to specify the location of the bids directory (`bids_dir`) and the inputs we want to parse for the workflow (`pybids_inputs`). The function returns a *`BidsDataset`*, which we'll assign to a variable called `inputs`:

Listing 14: Snakefile

```
from snakebids import bids, generate_inputs

configfile: 'config.yml'

inputs = generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
)

```

Note: Snakebids has transitioned to a new format for *generate_inputs()*. To get access to the old dict style return, the `use_bids_inputs` parameter must be set to False. A tutorial for the old syntax can be found on the v0.5.0 docs.

The config variables we need pre-defined are as follows::

Listing 15: config.yml

```
bids_dir: 'data'

fwhm:
  - 5
  - 10
  - 15
  - 20

pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
      extension: '.nii.gz'
      datatype: 'func'
    wildcards:
      - subject
      - session
      - acquisition
      - task
      - run
```

The `pybids_inputs` dict defines what types of inputs the workflow can make use of (i.e. the top-level keys, `bold` in this case), and for each input, how to filter for them (i.e. the `filters` dict), and what BIDS entities to replace with wildcards in the snakemake workflow (i.e. the `wildcards` dict).

Note: The `filters` dict is passed directly to the `get()` function in pybids, and thus is quite customizable.

Note: Entries in the `wildcards` list do not have to be in your bids dataset, but if they are, then they will be converted into wildcards (i.e. `task-{task}`) if they are in the filenames. The names for these also correspond with pybids (e.g. `acquisition` maps to `acq`).

The *BidsDataset* class returned by *generate_inputs()* summarizes the wildcards found in your bids dataset and has parameters to plug those wildcards into your workflow. To investigate it, add a print statement following `generate_inputs(...)`:

Listing 16: Snakefile

```
from snakebids import bids, generate_inputs

configfile: 'config.yml'

inputs = generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
)

print(inputs)

```

Run the workflow:

```
$ snakemake -nq
BidsDataset({
    "bold": BidsComponent(
        name="bold",
        path="/path/to/data/sub-{subject}/func/sub-{subject}_task-{task}_run-{run}_bold.
↪nii.gz",
        zip_lists={
            "subject": ["001",  "001" ],
            "task":    ["rest", "rest"],
            "run":     ["1",    "2"   ],
        },
    ),
})
```

As you can see, *BidsDataset* is just a special kind of `dict`. Its keys refer to the names of the input types you specified in the config file (in `pybids_inputs`). You can test this by running `print(list(inputs.keys))`. Each value contains an object summarizing that input type. We refer to these input types, and the objects that describe them, as *BidsComponents*.

Each *BidsComponent* has three primary attributes. *.name* is the name of the component, this will be the same as the dictionary key in the dataset. *.path* is the generic path of the component. Note the wildcards: `{subject}`, `{task}`, and `{run}`. These wildcards can be substituted for values that will uniquely define each specific path. *.zip_lists* contains these unique values. It's a simple `dict` whose keys are bids entities and whose values are `lists` of entity-values. Note the tabular format that printed in your console: each of the columns of this "table" correspond to the entity-values of one specific file.

Notice that `inputs['bold'].path` is the same as the path we wrote under `in_bold:` in our `config.yaml` file in *step 4*. In fact, we can go ahead and replace `config['in_bold']` in our `Snakemake` file with `inputs['bold'].path` and delete `in_bold` from `config.yaml`.

Listing 17: Snakefile

```
rule smooth:
    input:
        inputs['bold'].path
```

(continues on next page)

(continued from previous page)

```
    params:
```

Step 7: using input wildcards

BidsComponent.path already grants us a lot of flexibility, but we can still do more! In addition to the three main attributes of *BidsComponents* already described, the class offers a number of special properties we can use in our workflows. First, we'll look at *BidsComponent.wildcards*. This is a dict that maps each entity to the brace-wrapped `{wildcards}` we specified in `pybids_config`. If you printed this value in our test workflow, it would look like this:

```
inputs['bold'].wildcards == {
    'subject': '{subject}',
    'task': '{task}',
    'run': '{run}'
}
```

This is super useful when combined with *bids()*, as we can use the keyword expansion `**inputs["<input_name>"].wildcards` to set all the wildcard parameters to the *bids()* function. Thus, we can make our workflow even more general, by replacing this:

Listing 18: Snakefile

```
rule smooth:
    input:
        inputs['bold'].path
    params:
        sigma = calc_sigma_from_fwhm
    output:
        bids(
            root='results',
            subject='{subject}',
            task='{task}',
            run='{run}',
            fwhm='{fwhm}',
            suffix='bold.nii.gz'
        )
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

with this:

Listing 19: Snakefile

```
rule smooth:
    input:
        inputs['bold'].path
    params:
        sigma = calc_sigma_from_fwhm
    output:
        bids(
            root='results',
            fwhm='{fwhm}',
            suffix='bold.nii.gz',
```

(continues on next page)

(continued from previous page)

```
            **inputs['bold'].wildcards
        )
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

This effectively ensures that any bids entities from the input filenames (that are listed as pybids wildcards) get carried over to the output filenames. Note that we still have the ability to add on additional entities, such as `fwhm` here, and set the root directory and suffix.

Finally, we can use our *`BidsComponents`* to easily expand over the entity values found in our dataset using *`BidsComponent.expand()`*. This method gets used instead of the snakemake `expand()` function:

Listing 20: Snakefile

```
rule all:
    input:
        inputs['bold'].expand(
            bids(
                root='results',
                fwhm='{fwhm}',
                suffix='bold.nii.gz',
                **inputs['bold'].wildcards
            ),
            fwhm=config['fwhm'],
        )


```

Note: *`BidsComponent.expand()`* still uses snakemake's `expand()` under the hood, but applies extra logic to ensure only entity groups *actually* found in your dataset are used. If need to expand over additional wildcards, just add them as keyword args. They'll expand over every possible combination, just like snakemake's `expand()`.

For reference, here is the updated config file and Snakefile after these changes:

Listing 21: config.yml

```
bids_dir: 'data'

fwhm:
  - 5
  - 10
  - 15
  - 20

pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
      extension: '.nii.gz'
      datatype: 'func'
    wildcards:
      - subject
```

(continues on next page)

(continued from previous page)

```
      - session
      - acquisition
      - task
      - run
```

Listing 22: Snakefile

```
from snakebids import bids, generate_inputs

configfile: 'config.yml'

inputs = generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
)

rule all:
    input:
        inputs['bold'].expand(
            bids(
                root='results',
                fwhm='{fwhm}',
                suffix='bold.nii.gz',
                **inputs['bold'].wildcards
            ),
            fwhm=config['fwhm'],
        )


def calc_sigma_from_fwhm(wildcards):
    return f'{float(wildcards.fwhm)/2.355:0.2f}'


rule smooth:
    input:
        inputs['bold'].path
    params:
        sigma = calc_sigma_from_fwhm
    output:
        bids(
            root='results',
            fwhm='{fwhm}',
            suffix='bold.nii.gz',
            **inputs['bold'].wildcards
        )
    shell:
        'fslmaths {input} -s {params.sigma} {output}'
```

Step 8: creating a command-line executable

Now that we have pybids parsing to dynamically configure our workflow inputs based on our BIDS dataset, we are ready to turn our workflow into a BIDS App. BIDS Apps are command-line apps with a standardized interface (e.g. three required positional arguments: `bids_directory`, `output_directory`, and `analysis_level`).

We do this in snakebids by creating a python script containing a *`bidsapp.app`* built with the Snakemake integration plugin *`SnakemakeBidsApp`*. An example of this `run.py` script is shown below.

Listing 23: run.py

```
#!/usr/bin/env python3
from pathlib import Path

from snakebids import bidsapp, plugins

app = bidsapp.app(
    [
        plugins.SnakemakeBidsApp(Path(__file__).resolve().parent),
    ]
)

if __name__ == "__main__":
    app.run()
```

This creates a bidsapp with all the standard arguments. The usage will look something like this:

```
./run.py INPUT_DATASET OUTPUT_DATASET [participant|group] [--derivatives] [--participant-
↪label LABEL...]
```

A more complete description of bidsapp usage can be found at *Running Snakebids*.

Additional argument can be added using the `config.yml`. For instance, we can turn our `fwhm` setting into a CLI parameter with the following:

Listing 24: config.yml

```
parse_args:
  --fwhm:
    help: >
      Set the full-width-half-maximum values that should be used for smoothing.
    type: int
    nargs: +
    default:
      - 5
      - 10
      - 15
      - 20
```

Snakebids uses the `argparse` module, and each entry in this `parse_args` dict becomes a call to `add_argument()` from `argparse.ArgumentParser`. When you run the workflow, snakebids adds the named argument values to the config dict, so your workflow can make use of it as if you had manually added the variable to your configfile. See *parse_args* for more details.

In the above example, snakebids will automatically insert a key called `fwhm` into the snakemake `config` containing the values provided from the command line (or the default, if no values are provided).

The `analysis_level` positional argument can also be modified in the config. The available levels are in an `analysis_levels` list in the config. Specific snakemake targets can be mapped to these levels in the `targets_by_analysis_level` dict:

Listing 25: config.yml

```yaml
analysis_levels:
 - participant

targets_by_analysis_level:
  participant:
    - ''  # if '', then the first rule is run
```

Note: since we specified a `''` for the target rule, no target rule will be specified, so snakemake will just default to the first rule in the workflow.

Make the `run.py` script executable (`chmod a+x run.py`) and try running it now.

The updated configfile is here (the Snakefile did not change in this step):

Listing 26: config.yml

```yaml
bids_dir: 'data'


pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
      extension: '.nii.gz'
      datatype: 'func'
    wildcards:
      - subject
      - session
      - acquisition
      - task
      - run

analysis_levels:
 - participant

targets_by_analysis_level:
  participant:
    - ''  # if '', then the first rule is run

parse_args:
  --fwhm:
    help: >
      Set the full-width-half-maximum values that should be used for smoothing.
    type: int
    nargs: +
    default:
      - 5
      - 10
      - 15
```

(continues on next page)

(continued from previous page)

```
      - 20
```

7.3 Bids Function

The `bids` function generates a BIDS-like filepath corresponding to its keyword arguments. The generated filepath has the form:

```
[root]/[sub-{subject}]/[ses-{session]/[prefix]_[sub-{subject}]_[ses-{session}]_[{key}-
↪{val}_ ... ]_[suffix]
```

Use cases of the `bids` function include, at simplest, replacing a hard-coded BIDS file with an invocation of the `bids` function, so

```
"data/sub-01/ses-01/func/sub-01_ses-01_task-rest_acq-01_run-1_bold.nii.gz"
```

could become

```
bids(
    root="data",
    subject="01",
    session="01",
    datatype="func",
    task="rest",
    acq="01",
    run="1",
    suffix="bold.nii.gz"
)
```

If you wanted to specify that a rule should run on a BIDS file from any subject, session, acquisition, task, and run, you could change those keyword arguments to be snakemake wildcards:

```
bids(
    root="data",
    subject="{subject}",
    session="{session}",
    datatype="func",
    task="{task}",
    acq="{acq}",
    run="{run}",
    suffix="bold.nii.gz"
)
```

Using the subject and session keywords as wildcards is common enough that snakebids pre-populates a config variable (`subj_wildcards`) with these wildcards, allowing the `bids` call to look like the following:

```
bids(
    root="data",
    datatype="func",
    task="{task}",
    acq="{acq}",
    run="{run}",
```

(continues on next page)

(continued from previous page)

```
    suffix="bold.nii.gz",
    **inputs.subj_wildcards
)
```

Now if you want to process all inputs of a given form regardless of how their wildcards resolve, you can use *BidsComponent.expand* to expand over the entity-values found in your input dataset. As an example, to specify the output of a rule that preprocesses BOLD images (as specified in the example configuration), the following would resolve the subject, session, acquisition, task, and run wildcards:

```
inputs["bold"].expand(
    bids(
        root="output",
        datatype="func",
        desc="preproc",
        acq="{acq}",
        task="{task}",
        run="{run}",
        **inputs.subj_wildcards
    ),
)
```

7.3.1 Specs

The structure of the built path is based on the currently active BIDS spec. More information can be found on the *specs* page.

7.4 Bids Apps

7.4.1 Configuration

Snakebids is configured with a YAML (or JSON) file that extends the standard snakemake config file with variables that snakebids uses to parse an input BIDS dataset and expose the snakebids workflow to the command line.

Config Variables

pybids_inputs

A dictionary that describes each type of input you want to grab from an input BIDS dataset. Snakebids will parse your dataset with *generate_inputs()*, converting each input type into a *BidsComponent*. The value of each item should be a dictionary with keys `filters` and `wildcards`.

Filters

The value of `filters` should be a dictionary where each key corresponds to a BIDS entity, and the value specifies which values of that entity should be grabbed. The dictionary for each input is sent to the PyBIDS' `get()` function . `filters` can be set according to a few different formats:

- `string`: specifies an exact value for the entity. In the following example:

```
pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
      extension: '.nii.gz'
      datatype: 'func'
```

the bold component would match any paths under the `func/` datatype folder, with the suffix `bold` and the extension `.nii.gz`.

```
sub-xxx/.../func/sub-xxx_ses-xxx_..._bold.nii.gz
```

- `boolean`: constrains presence or absence of the entity without restricting its value. `False` requires that the entity be **absent**, while `True` requires the entity to be **present**, regardless of value.

```
pybids_inputs:
  derivs:
    filters:
      datatype: 'func'
      desc: True
      acquisition: False
```

The above example selects all paths in the `func/` datatype folder that have a `_desc-` entity but do not have the `_acq-` entity.

- `list`: Specify multiple string or boolean filters. Any path matching any one of the filters will be selected. Using `False` as one of the filters allows the entity to optionally be absent in addition to matching one of the string filters. Using `True` along with text is redundant, as `True` will cause any value to be selected. Using `True` with `False` is equivalent to not providing the filter at all.

These filters:

```
pybids_inputs:
  derivs:
    filters:
      acquisition:
        - False
        - MPRAGE
        - MP2RAGE
```

would select all of the following paths:

```
sub-001/ses-1/anat/sub-001_ses-001_acq-MPRAGE_run-1_T1w.nii.gz
sub-001/ses-1/anat/sub-001_ses-001_acq-MP2RAGE_run-1_T1w.nii.gz
sub-001/ses-1/anat/sub-001_ses-001_run-1_T1w.nii.gz
```

- To use regex for filtering, use an additional subkey set either to `match` or `search`, depending on which regex method you wish to use. This key may be set to any one of the above items (`str`, `bool`, or `list`). Only one such

key may be used.

These filters:

```yaml
pybids_inputs:
  derivs:
    filters:
      suffix:
        search: '[Tt]1'
      acquisition:
        match: MP2?RAGE
```

would select all of the following paths:

```
sub-001/ses-1/anat/sub-001_ses-001_acq-MPRAGE_run-1_T1.nii.gz
sub-001/ses-1/anat/sub-001_ses-001_acq-MP2RAGE_run-1_t1w.nii.gz
sub-001/ses-1/anat/sub-001_ses-001_acq-MPRAGE_run-1_qT1w.nii.gz
```

Note: `match` and `search` are both *filtering methods*. In addition to these, `get` is also a valid filtering method and may be used as the subkey for a filter. However, this is equivalent to directly providing the desired filter without a subkey:

```yaml
pybids_inputs:
  derivs:
    filters:
      suffix:
        get: T1w

# is the same as
pybids_inputs:
  derivs:
    filters:
      suffix: T1w
```

In other words, `get` is the default filtering method.

Wildcards

The value of `wildcards` should be a list of BIDS entities. Snakebids collects the values of any entities specified and saves them in the *`entities`* and *`zip_lists`* entries of the corresponding *`BidsComponent`*. In other words, these are the entities to be preserved in output paths derived from the input being described. Placing an entity in `wildcards` does not require the entity be present. If an entity is not found, it will be left out of *`entities`*. To require the presence of an entity, place it under `filters` set to `true`.

In the following (YAML-formatted) example, the `bold` input type is specified. BIDS files with the datatype `func`, suffix `bold`, and extension `.nii.gz` will be grabbed, and the `subject`, `session`, `acquisition`, `task`, and `run` entities of those files will be left as wildcards. The `task` entity must be present, but there must not be any `desc`.

```yaml
pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
      extension: '.nii.gz'
```

(continues on next page)

(continued from previous page)

```
      datatype: 'func'
      task: true
      desc: false
    wildcards:
      - subject
      - session
      - acquisition
      - task
      - run
```

pybidsdb_dir

PyBIDS allows for the use of a cached layout to be used in order to reduce the time required to index a BIDS dataset. A path (if provided) to save the *pybids* layout. If `None` or `''` is provided, the layout is not saved or used. The path provided must be absolute, otherwise the database will not be used.

pybidsdb_reset

A boolean determining whether the existing layout should be be updated. Default behaviour does not update the existing database if one is used.

analysis_levels

A list of analysis levels in the BIDS app. Typically, this will include participant and/or group. Note that the default (YAML) configuration file expects this mapping to be identified with the anchor `analysis_levels` to be aliased by `parse_args`.

targets_by_analysis_level

A mapping from the name of each `analysis_level` to the list of rules or files to be run for that analysis level.

parse_args

A dictionary of command-line parameters to make available as part of the BIDS app. Each item of the mapping is passed to argparse's `add_argument` function. A number of default entries are present in a new snakebids project's config file that structure the BIDS app's CLI, but additional command-line arguments can be added as necessary.

As in `ArgumentParser.add_argument()`, `type` may be used to convert the argument to the specified type. It may be set to any type that can be serialized into yaml, for instance, `str`, `int`, `float`, and `boolean`.

```
parse_args:
  --a-string:
    help: args are string by default
  --a-path:
    help: |
      A path pointing to data needed for the pipeline. These are still converted
      into strings, but are first resolved into absolute paths (see below)
```

(continues on next page)

(continued from previous page)

```
    type: Path
  --another-path:
    help: This type annotation does the same thing as above
    type: pathlib.Path
  --a-number:
    help: A number important for the analysis
    type: float
```

When CLI parameters are used to collect paths, `type` should be set to `Path` (or `pathlib.Path`). These arguments will still be serialized as strings (since yaml doesn't have a path type), but snakebids will automatically resolve all arguments into absolute paths. This is important to prevent issues with snakebids and relative paths.

debug

A boolean that determines whether debug statements are printed during parsing. Should be disabled (False) if you're generating DAG visualization with snakemake.

derivatives

A boolean (or path(s) to derivatives datasets) that determines whether snakebids will search in the derivatives subdirectory of the input dataset.

7.4.2 Workflows

Snakebids workflows are constructed the same way as any other Snakemake workflows, but with a few additions that make it easier to work with BIDS datasets.

To get access to these additions, the base Snakefile for a snakebids workflow should begin with the following boilerplate:

```
import snakebids
from snakebids import bids

configfile: 'config/snakebids.yml'

# Get input wildcards
inputs = snakebids.generate_inputs(
    bids_dir=config["bids_dir"],
    pybids_inputs=config["pybids_inputs"],
    pybidsdb_dir=config.get("pybidsdb_dir"),
    pybidsdb_reset=config.get("pybidsdb_reset"),
    derivatives=config.get("derivatives"),
    participant_label=config.get("participant_label"),
    exclude_participant_label=config.get("exclude_participant_label"),
)

```

Snakebids workflow features

The *snakebids.bids* function generates a properly-formatted BIDS filename with the specified entities, as documented in more detail elsewhere in this documentation.

snakebids.generate_inputs returns an instance of *snakebids.BidsDataset*, a special `dict` with keys mapping to the *BidsComponents* defined in *the config file*. Each *BidsComponent* contains a number of attributes to assist processing a BIDS dataset with snakemake. *generate_inputs()* should be called at the beginning of the workflow and assigned to a variable called `inputs`.

The *path* member of *BidsComponent* is generated by snakebids and contains a list of matched files for every input type. Often, the first rule to be invoked will use one or more entries in `inputs.path` as the input file specification.

The *expand()* method of *BidsComponent* is used to expand over files using only the entity-values found in your dataset. A usage pattern is as follows:

```
inputs["bold"].expand(
    bids(
        root="results",
        datatype="func",
        suffix="func.shape.gii",
        hemi="{hemi}",
        **inputs.wildcards["bold"]
    ),
    hemi=config["hemi"],
)
```

Extra entities provided to *BidsComponent.expand()* will expand the path across every possible combination of values, just like in the snakemake `expand()`.

By default, *BidsComponent.expand()* prevents partial expansion over paths, consistent with the default snakemake behaviour. To allow the presence of extra wildcards in the path, set the `allow_missing` argument in *BidsComponent.expand()* to `True`.

The *wildcards* member of *BidsComponent* is generated by snakebids and contains a dictionary mapping the wildcards for each input type to snakemake-formatted wildcards, for convenient use in the `bids` function.

Accessing the underlying *pybids* dataset

In addition to mapping all of the *BidsComponents* to their names, *BidsDataset* also has a *layout* member which gives access to the underlying `BIDSLayout`. This can be used to access advanced pybids features not covered by `snakebids`. Note that if `custom_paths` are specified for every *BidsComponent*, pybids indexing will be skipped and *layout* will be set to `None`. If your workflow relies on accessing this *layout*, you must ensure your users do not provide a `custom_path` for every single component, either in the config file or *via the CLI* (`--path_{component}`).

7.4.3 Plugins

Plugins allow you to extend the functionality of your *BIDS app* before and after parsing CLI arguments. For example, you can use a plugin to perform BIDS validation of your Snakebids app's input, which ensures your app is only executed if the input dataset is valid. You can either use those that are distributed with Snakebids (see *Using plugins*) or create your own plugins (see *Creating plugins*).

Note: For a full list of plugins distributed with Snakebids, see the *Plugins reference* page.

Nearly all of the functionality provided by *snakebids.bidsapp* is provided by plugins, including *SnakemakeBidsApp*.

Unlike in libraries such as pytest, plugins must be explicitly enabled to be included in the app. Installing them with pip is not enough! This affords a great deal of control over how the BIDS app is executed.

Using plugins

To add plugins to your *bidsapp*, pass them to the *SnakeBidsApp* constructor via the `plugins` parameter. Plugins are executed in LIFO order (last in, first out).

As an example, the *BidsValidator* plugin can be used to run the BIDS Validator on the input directory like so:

```
from snakebids import bidsapp, plugins

bidsapp.app([
    plugins.SnakemakeBidsApp("path/to/snakebids/app"),
    plugins.BidsValidator,
]).run()
```

Dependencies

Some plugins depend on other plugins. These dependencies will be loaded even if not specified in *bidsapp.app*. If a dependency needs explicit configuration, however, they may still be safely provided in app initialization, as snakebids will prevent duplicate registration.

For example, *SnakemakeBidsApp* depends on *BidsArgs*, *CliConfig*, *ComponentEdit*, etc, but these plugins may still be specified to change their configuration. The order of specification will be respected (e.g. LIFO).

```
from snakebids import bidsapp, plugins

bidsapp.app([
    plugins.SnakemakeBidsApp("path/to/snakebids/app"),
    # specify the BidsArgs plugin to override the argument group
    plugins.BidsArgs(argument_group="MAIN"),
]).run()
```

Creating plugins

Plugins are implemented using `pluggy`, the plugin system used and maintained by pytest. Actions are executed in one of several hooks. These are called at specified times as the argument parser is built, arguments are parsed, and the config is formatted.

A plugin is a class or module with methods or functions wrapped with the snakebids plugin hook decorator: *snakebids.bidsapp.hookimpl*. The name of the function determines the stage of app initialization at which it will be called. Each recognized function name (known as specs) comes with a specified set of available arguments. Not all the available arguments need to be used, however, they must be given the correct name. The *API documentation* contains the complete list of available specs, their corresponding initialization stages, and the arguments they can access.

As an example, a simplified version of the bids-validator plugin that runs the BIDS Validator could be defined as follows:

```python
import argparse
import subprocess
from typing import Any

from snakebids import bidsapp

class BidsValidator:
    """Perform BIDS validation of dataset

    Parameters
    ----------
    app
        Snakebids application to be run
    """

    @bidsapp.hookimpl
    def add_cli_arguments(self, parser: argparse.ArgumentParser):
        parser.add_argument("--skip-validation", dest="plugins.validator.skip")

    @bidsapp.hookimpl
    def finalize_config(self, config: dict[str, Any]) -> None:
        # Skip bids validation
        if config["plugins.validator.skip"]:
            return

        try:
            subprocess.run(
                ["bids-validator", config["bids_dir"]], check=True
            )
        except subprocess.CalledProcessError as err:
            raise InvalidBidsError from err


class InvalidBidsError(SnakebidsPluginError):
    """Error raised if input BIDS dataset is invalid,
    inheriting from SnakebidsPluginError.
    """
```

In this example, two hooks were used. The *add_cli_arguments()* hook is called before CLI arguments are parsed. Here, it adds an argument allowing end users of our app to skip bids validation. Note that the *spec* specifies three arguments available to this hook (`parser`, `config`, and `argument_groups`), however, we only used `parser` here.

Note: When adding plugin-specific parameters to the config dictionary, it is recommended to use namespaced keys (e.g. `plugins.validator.skip`). This will help ensure plugin-specific parameters do not conflict with other parameters already defined in the dictionary or by other plugins.

The *finalize_config()* hook is called after `config` is updated with the results of argument parsing. In our plugin, this is where validation is actually performed. Note how the argument added in the previous hook is now read from `config`. Any modifications made to `config` will be carried forward into the app's remaining lifetime.

A plugin can be used to implement any logic that can be handled by a Python function. In the above example, you may also want to add some logic to check if the BIDS Validator is installed and pass along a custom error message if it is not. Created plugins can then be used within a Snakebids workflow, similar to the example provided in *Using plugins*

section. Prospective plugin developers can take a look at the source of the `snakebids.plugins` module for examples.

Note: When creating a custom error for your Snakebids plugin, it is recommended to inherit from *SnakebidsPluginError* such that errors will be recognized as a plugin error.

Specifying dependencies

Dependencies may be specified using a `DEPENDENCIES` attribute in your plugin class or module. It should be set to a tuple containing fully initialized plugin references. These dependencies will be registered after the depending plugin and therefore run first (due to pluggy's LIFO order).

```
from snakebids import bidsapp, plugins

class MyPlugin:
    DEPENDENCIES = (
        plugins.CliConfig(),
        plugins.BidsArgs(),
    )

    @bidsapp.hookimpl
    def finalize_config(self, config):
        ...
```

Snakebids apps rely on a configuration file (`snakebids.yml`). This file specifies which files from a BIDS dataset should be used as input. The apps also utilize workflow definitions, which are written in one or more Snakefile(s) and determine how the input files are processed.

Note: For an easy setup of new Snakebids apps with convenient command-line functions, we recommend installing Snakebids using `pipx`. Visit the following page for instructions on how to install `pipx`.

Once Snakebids is installed, you can generate a customized Snakebids project by running the command `snakebids create` and providing the necessary information when prompted.

7.5 Running Snakebids

Once you've specified a snakebids app with a config file and one or more workflow files, you're ready to invoke your snakebids app with the standard BIDS app CLI.

Snakebids apps generated with the cookiecutter template will have a simple executable called `run.py`, which exposes the BIDS app CLI, with any additional options configured in the `snakebids.yml` config file. Installing the project with pip will also add an executable with the project's name to the path. Any Snakemake arguments should be added to the end of the invocation.

While Snakebids apps use the standard BIDS app CLI (i.e. `{app_name} {input} {output} {analysis_level}`), it is possible to override the input location for each input type defined in the configuration file. By passing the path override argument `--path_{input_type} {path}`, where `{input_type}` is the name of an input type defined in the configuration file, and `{path}` is the path to a directly containing files appropriate for that input type. If a path override argument is provided for every input type, an `{input}` argument must still be provided to the BIDS app CLI, but it does not need to be an existing directory; a string like `-` will work.

Note that if any rules in the Snakebids workflow use Singularity containers, special precautions must be taken to ensure the input dataset is bound to the Singularity environment. Either:

1. Inputs are copied into a working subdirectory of the output directory before any processing that requires a Singularity container is performed, or:
2. The `SINGULARITY_BINDPATH` environment variable binds the location of the input dataset.

Indexing of large datasets can be a time-consuming process. Leveraging the functionality of `PyBIDS`, Snakebids offers a convenient solution by allowing you to create or utilize an existing database. With this approach, the indexing of datasets is only performed when explicitly requested, typically following changes to the dataset. To create or use an existing database, you can invoke the following CLI arguments:

1. `--pybidsdb-dir {dir}`: specify the path to the database directory
2. `--pybidsdb-reset`: indicate that an existing database should be updated

The boilerplate app starts with the validator plugin enabled - without it, validation is not performed. By default, this feature uses the command-line (node.js) version of the validator. If this is not found to be installed on the system, the `pybids` version of validation will be performed instead. To opt-out of validation, invoke the `--skip-bids-validation` flag. Details related to using and creating plugins can be found on the *plugins* page.

7.5.1 Workflow mode

Snakebids apps use a BIDS app CLI, giving great flexibility when switching datasets. However, when developing a Snakebids app or when running the app repeatedly on the same dataset, it can be more convenient to directly call the Snakemake CLI. Snakebids facilitates this using workflow mode.

Workflow mode activates when the Snakebids app itself is used as the output folder. Snakebids will save your config file in the config folder and put any outputs in the results folder. After the first Snakebids call, the Snakemake CLI can be called directly using the generated config file.

As an example, suppose we have a BIDS formatted dataset:

```
dataset
├── code
├── dataset_description.json
├── derivatives
├── sub-001
├── sub-002
├── sub-003
├── sub-004
├── sub-005
├── sub-006
├── sub-007
├── sub-008
└── sub-009
```

We'd like to develop a new processing pipeline called SuperCorrect. We'll start by making a new directory in `derivatives` using the Snakemake CookieCutter template:

```
dataset
├── code
├── dataset_description.json
├── derivatives
│   └── super_correct
│       ├── config
```

(continues on next page)

(continued from previous page)

```
│       │   └── snakebids.yml
│       ├── pipeline_description.json
│       ├── run.py
│       └── workflow
│           └── Snakefile
├── sub-001
├── sub-002
├── sub-003
├── sub-004
├── sub-005
├── sub-006
├── sub-007
├── sub-008
└── sub-009
```

cd to `super_correct`:

```
super_correct
├── config
│   └── snakebids.yml
├── pipeline_description.json
├── run.py
└── workflow
    └── Snakefile
```

We then develop our pipeline, writing our `Snakefile`, rules, etc. When we're ready to start testing, we start by calling our `run.py` function:

```
run.py ../../ . participant [snakemake args]
```

We'll see a message telling us the app is running in snakemake mode and, if our workflow doesn't have any bugs, the app will run! `config/snakebids.yml` will be updated to include all the information we passed into the CLI. Output files will be in the `results` folder:

```
super_correct
├── config
│   └── snakebids.yml
├── pipeline_description.json
├── results
│   └── super_correct
│       ├── dataset_description.json
│       ├── sub-001
│       ├── sub-002
│       ├── sub-003
│       ├── sub-004
│       ├── sub-005
│       ├── sub-006
│       ├── sub-007
│       ├── sub-008
│       └── sub-009
├── run.py
└── workflow
    └── Snakefile
```

From now on, instead of calling `run.py`, we can just the Snakemake CLI directly. It will use the same inputs and outputs saved into our config by Snakebids:

```
snakemake [args]
```

You can still use the Snakebids CLI on other datasets. However, if you plan on modifying any files, including config, to make the Snakebids app suitable for the new dataset, it's recommended to use git to clone the app into the `derivatives` folder of the new dataset. Alternatively, you can call `run.py` with the `--workflow-mode` flag:

```
run.py /path/to/newdata /path/to/newdata/derivatives/super_correct participant --
↪workflow-mode [snakemake args]
```

This will make a copy of the Snakebids app at the new output directory, excluding the `results` folder and some configuration folders/files (`.snakemake/`, `.snakebids`). It will, again, make a new config file, and put new results in the `output/results` folder.

7.6 Migrations

Snakebids has rapidly evolved over the last few versions, resulting in a number of breaking changes. Use the following guides if you're migrating codes from old snakemake versions.

7.6.1 0.5 to 0.8+

Note: Be sure to also *migrate* your `run.py` file to the new snakebids 0.12 syntax!

Starting in version 0.8, *snakebids.generate_inputs()* returns a *BidsInputs* object instead of a *dict*. This requires a change in the way info is accessed. The previous *dict* had top-level keys such as `"input_lists"`. After selecting such a key, you would pass the name of a component to get the information sought:

```
config.update(snakebids.generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
    use_bids_inputs=False,
))

config["input_lists"]["t1w"]
```

Now, the components are top level keys, and the type of property being requested is accessed using an attribute:

```
inputs = snakebids.generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
)

inputs["t1w"].entities
```

Note that the old behaviour can still be achieved by setting `use_bids_inputs=False`, as shown in the above example. However, we encourage all users to upgrade to take advantage of all the new features Snakebids has to offer.

1. Assign `generate_inputs()` to a variable called `inputs`

Because *`generate_inputs()`* no longer returns a dict, you cannot use it to update `config`, as was previously recommended. The new best practice is to assign its return to a variable called `inputs`:

```
inputs = snakebids.generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
)
```

2. Change references to `config`

All references to `config['<attr>']['<comp>']`, where `<attr>` is one of `'input_path'`, `'input_zip_lists'`, `'input_lists'`, or `'input_wildcards'`, must be updated to `input['<comp>'].<attr>`. The following regexes may be helpful for search and replace:

```
# match
config\[([\x22\x27])(input_path|input_zip_lists|input_lists|input_wildcards)\1\]\[([\x22\
↪x27])(\w+)\3\]

# replace
inputs["\4"].\2
```

In addition, all references to `config['<attr>']` where `<attr>` is one of `'sessions'`, `'subjects'`, or `'subj_wildcards'` must be updated to `input.<attr>`. The following regexes may be helpful:

```
# match
config\[([\x22\x27])(sessions|subjects|subj_wildcards)\1\]

# replace
inputs.\2
```

3. Update attribute names into modern forms

Although the previous attribute names are being kept around as aliases, we recommend you update to the more modern, sleeker equivalents. Replacements should be made according to the following table:

- `input_path` -> *`path`*
- `input_lists` -> *`entities`*
- `input_zip_lists` -> *`zip_lists`*
- `input_wildcards` -> *`wildcards`*

4. Switch to `expand()` method

Calls to snakemake's `expand()` should be replaced with the new *`expand()`* method available on *`BidsComponent`*. See *the section* in 0.7-0.8 migration guide for more details.

7.6.2 0.7 to 0.8+

Warning: If your code still has bits like this:

```
config.update(generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
))
```

Check out the *pre-0.6 migration guide* to a guide on how to upgrade!

Note: Be sure to also *migrate* your `run.py` file to the new snakebids 0.12 syntax!

Default return of `generate_inputs()`

V0.8 switches the default return value of *`generate_inputs()`* from *`BidsDatasetDict`* to *`BidsDataset`*. Legacy code still relying on the old dictionary can avoid the update by setting the `use_bids_inputs` pararmeter in *`generate_inputs()`* to `False`:

```
config.update(generate_inputs(
    bids_dir=config['bids_dir'],
    pybids_inputs=config['pybids_inputs'],
    use_bids_inputs=False,
))
```

Code that previously set `use_bids_inputs=True` should remove that line from *`generate_inputs()`*. Such manual assignment is deprecated.

Properties of `BidsDataset`

The behaviour of the properties of *`BidsDataset`*, including *`path`*, *`zip_lists`*, *`entities`*, and *`wildcards`* is set to change in an upcoming release, thus, their current use is deprecated. Code should now access these properties via the *`BidsComponent`*. For instance:

```
# deprecated in v0.8
inputs.wildcards["t1w"]
inputs.entities["t1w"]

# should now use
inputs["t1w"].wildcards
inputs["t1w"].entities
```

New `expand()` method

V0.8 features a new *`expand()`* method on *`BidsComponent`*. This method automatically ensures only entity-values actually contained in your dataset are used when expanding over a path. It supports the addition of extra wildcards, and can expand over the component *`path`* or any number of provided paths. It should generally be preferred over snakemake's `expand()` when *`BidsComponents`* are involved, due to the increased safety and ease of use.

An `expand` call that used to look like this:

```
rule all:
    input:
        expand(
            expand(
                bids(
                    root=root,
                    desc="{smooth}",
                    **inputs["bold"].wildcards,
                ),
                allow_missing=True,
                smooth=[1, 2, 3, 4],
            ),
            zip,
            **inputs["bold"].zip_lists,
        )
```

can now be written like this:

```
rule all:
    input:
        inputs['bold'].expand(
            bids(
                root=root,
                desc="{smooth}",
                **inputs["bold"].wildcards,
            ),
            smooth=[1, 2, 3, 4],
        )
```

7.6.3 0.11 to 0.12+

Snakebids 0.12 introduces a *new, more flexible module* for creating bidsapps. This affects the syntax of the `run.py` file. Older versions used the *`snakebids.app.SnakeBidsApp`* class to initialize the bidsapp, and this method will still work for the forseeable future. Switching to the new syntax will give access to new plugins and integrations and ensure long term support.

If you haven't heavily modified your `run.py` file, you can transition simply by replacing it with the following:

```
#!/usr/bin/env python3
from pathlib import Path

from snakebids import bidsapp, plugins

app = bidsapp.app(
```

(continues on next page)

(continued from previous page)

```
    [
        plugins.SnakemakeBidsApp(Path(__file__).resolve().parent),
        plugins.BidsValidator(),
        plugins.Version(distribution="<app_name_here>"),
    ]
)


def get_parser():
    """Exposes parser for sphinx doc generation, cwd is the docs dir."""
    return app.build_parser().parser


if __name__ == "__main__":
    app.run()
```

The snakemake workflow will work in exactly the same way.

7.7 API

7.7.1 Path Building

snakebids.**bids**(*root=None*, ***, *datatype=None*, *prefix=None*, *suffix=None*, *extension=None*, ***entities*)

Generate bids or bids-like paths.

File path is of the form:

```
[root]/[sub-{subject}]/[ses-{session]/
[prefix]_[sub-{subject}]_[ses-{session}]_[{key}-{val}_ ... ]_[suffix]
```

If no arguments are specified, an empty string will be returned.

Datatype and prefix may not be used in isolation, but must be given with another entity.

BIDS paths are built based on specs, which are versioned for long-term stability. The latest version is `v0_0_0`. Information on its spec can be found at *v0_0_0()*.

Warning: By default, `bids()` will always use the latest BIDS spec. This is unsafe for production environments, as the spec may be updated without warning, even on patch releases. These updates may change the path output by `bids()`, resulting in breaking changes in downstream apps

Production code should always explicitly set the spec version using *set_bids_spec()*:

```
from snakebids import set_bids_spec
set_bids_spec("v0_0_0")
```

Parameters

- **root** (*str | Path | None*) – Root folder to include in the path (e.g. `results`)
- **datatype** (*str | None*) – Folder to include after sub-/ses- (e.g. `anat`, `dwi`)

- **prefix** (*str | None*) – String to prepend to the file name. Useful for injecting custom entities at the front of the filename, e.g. `tpl-{tpl}`
- **suffix** (*str | None*) – Suffix plus, optionally, the extension (e.g. `T1w.nii.gz`)
- **extension** (*str | None*) – bids extension, beginning with `.` (e.g. `.nii.gz`). Typically shouldn't be specified manually: extensions should be listed along with the suffix.
- **entities** (*str | bool*) – bids entities as keyword arguments paired with values (e.g. `space="T1w"` for `space-T1w`)

Return type
str

Examples

Below is a rule using bids naming for input and output:

```
rule proc_img:
    input: 'sub-{subject}_T1w.nii.gz' output:
    'sub-{subject}_space-snsx32_desc-preproc_T1w.nii.gz'
```

With bids() you can instead use:

```
rule proc_img: input: bids(subject='{subject}',suffix='T1w.nii.gz')
output: bids(
    subject='{subject}', space='snsx32', desc='preproc',
    suffix='T1w.nii.gz'
)
```

Note that here we are not actually using "functions as inputs" in snakemake, which would require a function definition with wildcards as the argument, and restrict to input/params, but bids() is being used simply to return a string.

Also note that space, desc and suffix are NOT wildcards here, only {subject} is. This makes it easy to combine wildcards and non-wildcards with bids-like naming.

However, you can still use bids() in a lambda function. This is especially useful if your wildcards are named the same as bids entities (e.g. {subject}, {session}, {task} etc..):

```
rule proc_img:
    input: lambda wildcards: bids(**wildcards,suffix='T1w.nii.gz') output:
    bids(
        subject='{subject}', space='snsx32', desc='preproc',
        suffix='T1w.nii.gz'
    )
```

Or another example where you may have many bids-like wildcards used in your workflow:

```
rule denoise_func:
    input: lambda wildcards: bids(**wildcards, suffix='bold.nii.gz') output:
    bids(
        subject='{subject}', session='{session}', task='{task}',
        acq='{acq}', desc='denoise', suffix='bold.nii.gz'
    )
```

In this example, all the wildcards will be determined from the output and passed on to bids() for inputs. The output filename will have a 'desc-denoise' flag added to it.

Also note that even if you supply entities in a different order, the entities will be ordered based on the OrderedDict defined here. If entities not known are provided, they will be just be placed at the end (before the suffix), in the order you provide them in.

snakebids.**bids_factory**(*spec*)

Generate bids functions according to the supplied spec.

Parameters

- **spec** (*List[BidsPathEntitySpec]*) – Valid Bids Spec object
- **_implicit** (*bool*) – Flag used internally to mark the default generated bids function. The resulting builder will warn when custom entities are used

Return type
BidsFunction

snakebids.**set_bids_spec**(*spec*)

Set the spec to be used by path generation functions (such as *bids()*).

Parameters
spec (BidsPathSpec | *VALID_SPECS*) – Either a spec object, or the name of a builtin *spec*

Specs

BIDS specs control the formatting of paths produced by the *bids()* function. They specify the order of recognized entities, placing `ses-X` after `sub-Y`, for instance, no matter what order they are specified in the function. Unrecognized entitites are placed in the order specified in the function call.

Because of this, each addition of entities to the spec presents a potentially breaking change. Suppose an entity called `foo` were added to the spec. Calls to *bids()* with `foo` as an argument would place the entity at the end of the path:

```
from snakebids import bids

# Before foo is in the spec
bids(
    subject="001",
    session="1",
    label="WM",
    foo="bar",
    suffix="data.nii.gz",
) == "sub-001_ses-1_label-WM_foo-bar_data.nii.gz"
```

The addition of `foo` to the spec might move the position of the entity forward in the output:

```
# After foo is in the spec
bids(
    subject="001",
    session="1",
    label="WM",
    foo="bar",
    suffix="data.nii.gz",
) == "sub-001_ses-1_foo-bar_label-WM_data.nii.gz"
```

This would change the output paths of workflow using this function call, causing a breaking change in workflow behaviour.

To ensure stable path generation across releases, Snakebids ships with versioned specs that can be explicitly set using `snakebids.set_bids_specs()`. These specs are named after the snakebids version they release with. By default, *bids()* will always use the latest spec, but production code should generally declare the spec to be used by the workflow:

```
from snakebids import set_bids_spec
set_bids_spec("v0_0_0")
```

This is especially true of workflows using custom entities. To emphasize this, a warning is issued in python scripts and apps using such entities without declaring a spec version.

snakebids.paths.specs.**v0_0_0**(*subject_dir=True*, *session_dir=True*)

Get the v0.0.0 BidsPathSpec.

This spec alone equips *bids()* with 2 extra arguments: `include_subject_dir` and `include_session_dir`. These default to `True`, but if set `False`, remove the subject and session dirs respectively from the output path. For future specs, this behaviour should be achieved by modifying the spec and generating a new *bids()* function

Formatted as:

```
sub-{subject}/ses-{session}/{datatype}/{prefix}_sub-{subject}_ses-{session}_
task-{task}_acq-{acq}_ce-{ce}_rec-{rec}_dir-{dir}_run-{run}_mod-{mod}_
echo-{echo}_hemi-{hemi}_space-{space}_res-{res}_den-{den}_label-{label}_
desc-{desc}_..._{suffix}{extension}
```

Parameters

- **subject_dir** (*bool*) – If False, downstream path generator will not include the subject dir *sub-{subject}/**
- **session_dir** (*bool, optional*) – If False, downstream path generator will not include the session dir **/ses-{session}/**

Return type
List[BidsPathEntitySpec]

snakebids.paths.specs.**v0_11_0**(*subject_dir=True*, *session_dir=True*)

Spec corresponding to BIDS v1.9.0.

Significantly expanded from the v0.0.0 spec, now including long names for every relevant entity. In addition to the official spec, it includes *from* and *to* entities intended for transformations. Unknown entities are placed just before desc, so that the description entity is always last.

Formatted as:

```
sub-{subject}/ses-{session}/{datatype}/{prefix}_sub-{subject}_ses-{session}_
sample-{sample}_task-{task}_tracksys-{tracksys}_acq-{acquisition}_
ce-{ceagent}_stain-{staining}_trc-{tracer}_rec-{reconstruction}_
dir-{direction}_run-{run}_mod-{modality}_echo-{echo}_flip-{flip}_
inv-{inversion}_mt-{mt}_proc-{processed}_part-{part}_space-{space}_
atlas-{atlas}_seg-{segmentation}_hemi-{hemisphere}_res-{resolution}_
den-{density}_roi-{roi}_from-{from}_to-{to}_split-{split}_
recording-{recording}_chunk-{chunk}_model-{model}_subset-{subset}_
label-{label}_..._desc-{description}_{suffix}{extension}
```

Parameters

- **subject_dir** (*bool*) – If False, downstream path generator will not include the subject dir *sub-{subject}/**
- **session_dir** (*bool, optional*) – If False, downstream path generator will not include the session dir */ses-{session}/**

Return type
List[BidsPathEntitySpec]

snakebids.paths.specs.**latest**()
Points to the most recent spec

Types

class snakebids.**BidsFunction**(**args*, ***kwargs*)
Signature for functions returned by `bids_factory`.

See *bids()* for more details

class snakebids.paths.**BidsPathEntitySpec**
Defines an entity in a bids path.

entity: *str*
Entity full name

tag: *str*
Short entity name, as appears in the path

dir: *bool*
If true, a directory with the entity-value pair is created

class snakebids.paths.**BidsPathSpec**

7.7.2 Dataset Creation

snakebids.**generate_inputs**(*bids_dir: Path | str*, *pybids_inputs:* InputsConfig, *pybidsdb_dir: Path | str | None = None*, *pybidsdb_reset: bool = None*, *derivatives: bool | Path | str = False*, *pybids_config: str | None = None*, *limit_to: Iterable[str] | None = None*, *participant_label: Iterable[str] | str | None = None*, *exclude_participant_label: Iterable[str] | str | None = None*, *use_bids_inputs: Literal[True] | None = None*, *index_metadata: bool = False*, *validate: bool = False*, *pybids_database_dir: Path | str | None = None*, *pybids_reset_database: bool = None*) → *BidsDataset*

snakebids.**generate_inputs**(*bids_dir: Path | str*, *pybids_inputs:* InputsConfig, *pybidsdb_dir: Path | str | None = None*, *pybidsdb_reset: bool = None*, *derivatives: bool | Path | str = False*, *pybids_config: str | None = None*, *limit_to: Iterable[str] | None = None*, *participant_label: Iterable[str] | str | None = None*, *exclude_participant_label: Iterable[str] | str | None = None*, *use_bids_inputs: Literal[False] = None*, *index_metadata: bool = False*, *validate: bool = False*, *pybids_database_dir: Path | str | None = None*, *pybids_reset_database: bool = None*) → *BidsDatasetDict*

Dynamically generate snakemake inputs using pybids_inputs.

Pybids is used to parse the bids_dir. Custom paths can also be parsed by including the custom_paths entry under the pybids_inputs descriptor.

Parameters

- **bids_dir** – Path to bids directory
- **pybids_inputs** – Configuration for bids inputs, with keys as the names (`str`)

  Nested *dicts* with the following required keys (for complete info, see *InputConfig*):

  - `"filters"`: Dictionary of entity: "values" (dict of str -> str or list of str). The entity keywords should the bids tags on which to filter. The values should be an acceptable str value for that entity, or a list of acceptable str values.
  - `"wildcards"`: List of (str) bids tags to include as wildcards in snakemake. At minimum this should usually include `['subject','session']`, plus any other wildcards that you may want to make use of in your snakemake workflow, or want to retain in the output paths. Any wildcards in this list that are not in the filename will just be ignored.
  - `"custom_path"`: Custom path to be parsed with wildcards wrapped in braces, as in `/path/to/sub-{subject}/{wildcard_1}-{wildcard_2}`. This path will be parsed without pybids, allowing the use of non-bids-compliant paths.
- **pybidsdb_dir** – Path to database directory. If None is provided, database is not used
- **pybidsdb_reset** – A boolean that determines whether to reset / overwrite existing database.
- **derivatives** – Indicates whether pybids should look for derivative datasets under bids_dir. These datasets must be properly formatted according to bids specs to be recognized. Defaults to False.
- **limit_to** – If provided, indicates which input descriptors from pybids_inputs should be parsed. For example, if pybids_inputs describes `"bold"` and `"dwi"` inputs, and `limit_to = ["bold"]`, only the "bold" inputs will be parsed. "dwi" will be ignored
- **participant_label** – Indicate one or more participants to be included from input parsing. This may cause errors if subject filters are also specified in pybids_inputs. It may not be specified if exclude_participant_label is specified
- **exclude_participant_label** – Indicate one or more participants to be excluded from input parsing. This may cause errors if subject filters are also specified in pybids_inputs. It may not be specified if participant_label is specified
- **use_bids_inputs** – If False, returns the classic *BidsDatasetDict* instead of :class`BidsDataset`. Setting to True is deprecated as of v0.8, as this is now the default behaviour
- **index_metadata** – If True indexes metadata of BIDS directory using pybids, otherwise skips indexing.
- **validate** – If True performs validation of BIDS directory using pybids, otherwise skips validation.

Returns

Object containing organized information about the bids inputs for consumption in snakemake. See the documentation of *BidsDataset* for details and examples.

Return type

BidsDataset | *BidsDatasetDict*

Example

As an example, consider the following BIDS dataset:

```
example
├── README.md
├── dataset_description.json
├── participant.tsv
├── sub-001
│   ├── ses-01
│   │   ├── anat
│   │   │   ├── sub-001_ses-01_run-01_T1w.json
│   │   │   ├── sub-001_ses-01_run-01_T1w.nii.gz
│   │   │   ├── sub-001_ses-01_run-02_T1w.json
│   │   │   └── sub-001_ses-01_run-02_T1w.nii.gz
│   │   └── func
│   │       ├── sub-001_ses-01_task-nback_bold.json
│   │       ├── sub-001_ses-01_task-nback_bold.nii.gz
│   │       ├── sub-001_ses-01_task-rest_bold.json
│   │       └── sub-001_ses-01_task-rest_bold.nii.gz
│   └── ses-02
│       ├── anat
│       │   ├── sub-001_ses-02_run-01_T1w.json
│       │   └── sub-001_ses-02_run-01_T1w.nii.gz
│       └── func
│           ├── sub-001_ses-02_task-nback_bold.json
│           ├── sub-001_ses-02_task-nback_bold.nii.gz
│           ├── sub-001_ses-02_task-rest_bold.json
│           └── sub-001_ses-02_task-rest_bold.nii.gz
└── sub-002
    ├── ses-01
    │   ├── anat
    │   │   ├── sub-002_ses-01_run-01_T1w.json
    │   │   ├── sub-002_ses-01_run-01_T1w.nii.gz
    │   │   ├── sub-002_ses-01_run-02_T1w.json
    │   │   └── sub-002_ses-01_run-02_T1w.nii.gz
    │   └── func
    │       ├── sub-002_ses-01_task-nback_bold.json
    │       ├── sub-002_ses-01_task-nback_bold.nii.gz
    │       ├── sub-002_ses-01_task-rest_bold.json
    │       └── sub-002_ses-01_task-rest_bold.nii.gz
    └── ses-02
        └── anat
            ├── sub-002_ses-02_run-01_T1w.json
            ├── sub-002_ses-02_run-01_T1w.nii.gz
            ├── sub-002_ses-02_run-02_T1w.json
            └── sub-002_ses-02_run-02_T1w.nii.gz
```

With the following `pybids_inputs` defined in the config file:

```
pybids_inputs:
  bold:
    filters:
      suffix: 'bold'
```

(continues on next page)

(continued from previous page)

```
      extension: '.nii.gz'
      datatype: 'func'
    wildcards:
      - subject
      - session
      - acquisition
      - task
      - run
```

Then `generate_inputs(bids_dir, pybids_input)` would return the following values:

```
BidsDataset({
    "bold": BidsComponent(
        name="bold",
        path="bids/sub-{subject}/ses-{session}/func/sub-{subject}_ses-{session}_
↪task-{task}_bold.nii.gz",
        zip_lists={
            "subject": ["001",   "001",  "001",   "001",  "002",   "002" ],
            "session": ["01",    "01",   "02",    "02",   "01",    "01"  ],
            "task":    ["nback", "rest", "nback", "rest", "nback", "rest"],
        },
    ),
    "t1w": BidsComponent(
        name="t1w",
        path="example/sub-{subject}/ses-{session}/anat/sub-{subject}_ses-{session}_
↪run-{run}_T1w.nii.gz",
        zip_lists={
            "subject": ["001", "001", "001", "002", "002", "002", "002"],
            "session": ["01",  "01",  "02",  "01",  "01",  "02",  "02" ],
            "run":     ["01",  "02",  "01",  "01",  "02",  "01",  "02" ],
        },
    ),
})
```

7.7.3 Dataset Manipulation

snakebids.**filter_list**(*zip_list*, *filters*, *return_indices_only=False*, *regex_search=False*)

Filter zip_list, including only entries with provided entity values.

Parameters

- **zip_list** (ZipListLike) – generated zip lists dict from config file to filter
- **filters** (*Mapping[str, Iterable[str] | str]*) – wildcard values to filter the zip lists
- **return_indices_only** (*bool*) – return the indices of the matching wildcards
- **regex_search** (*bool*) – Use regex matching to filter instead of the default equality check.

Return type

ZipList | list[int]

Examples

```
>>> import snakebids
```

Filtering to get all `subject='01'` scans:

```
>>> snakebids.filter_list(
...     {
...         'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
...         'acq': ['98','98','98','98','99','99','99','99'],
...         'subject': ['01','01','02','02','01','01','02','02' ]
...     },
...     {'subject': '01'}
... ) == {
...     'dir': ['AP', 'PA', 'AP', 'PA'],
...     'acq': ['98', '98', '99', '99'],
...     'subject': ['01', '01', '01', '01']
... }
True
```

Filtering to get all `acq='98'` scans:

```
>>> snakebids.filter_list(
...     {
...         'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
...         'acq': ['98','98','98','98','99','99','99','99'],
...         'subject': ['01','01','02','02','01','01','02','02' ]
...     },
...     {'acq': '98'}
... ) == {
...     'dir': ['AP', 'PA', 'AP', 'PA'],
...     'acq': ['98', '98', '98', '98'],
...     'subject': ['01', '01', '02', '02']
... }
True
```

Filtering to get all `dir=='AP'` scans:

```
>>> snakebids.filter_list(
...     {
...         'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
...         'acq': ['98','98','98','98','99','99','99','99'],
...         'subject': ['01','01','02','02','01','01','02','02' ]
...     },
...     {'dir': 'AP'}
... ) == {
...     'dir': ['AP', 'AP', 'AP', 'AP'],
...     'acq': ['98', '98', '99', '99'],
...     'subject': ['01', '02', '01', '02']
... }
True
```

Filtering to get all `subject='03'` scans (i.e. no matches):

```
>>> snakebids.filter_list(
...     {
...         'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
...         'acq': ['98','98','98','98','99','99','99','99'],
...         'subject': ['01','01','02','02','01','01','02','02' ]
...     },
...     {'subject': '03'}
... ) == {
...     'dir': [],
...     'acq': [],
...     'subject': []
... }
True
```

snakebids.**get_filtered_ziplist_index**(*zip_list*, *wildcards*, *subj_wildcards*)

Return the indices of all entries matching the filter query.

Parameters

- **zip_list** (*dict*) – lists for scans in a dataset, zipped to get each instance
- **wildcards** (*dict*) – wildcards for the single instance for querying it's index
- **subj_wildcards** (*dict*) – keys of this dictionary are used to pick out the subject/(session) from the wildcards

Return type

int | list[int]

Examples

```
>>> import snakebids
```

In this example, we have a dataset where with scans from two subjects, where each subject has `dir-AP` and `dir-PA` scans, along with `acq-98` and `acq-99`:

- `sub-01_acq-98_dir-AP_dwi.nii.gz`
- `sub-01_acq-98_dir-PA_dwi.nii.gz`
- `sub-01_acq-99_dir-AP_dwi.nii.gz`
- `sub-01_acq-99_dir-PA_dwi.nii.gz`
- `sub-02_acq-98_dir-AP_dwi.nii.gz`
- `sub-02_acq-98_dir-PA_dwi.nii.gz`
- `sub-02_acq-99_dir-AP_dwi.nii.gz`
- `sub-02_acq-99_dir-PA_dwi.nii.gz`

The `zip_list` produced by `generate_inputs()` is the set of entities that when zipped together, e.g. with `expand(path, zip, **zip_list)`, produces the entity combinations that refer to each scan:

```
{
    'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
    'acq': ['98','98','98','98','99','99','99','99'],
```

(continues on next page)

(continued from previous page)

```
    'subject': ['01','01','02','02','01','01','02','02']
}
```

The `filter_list()` function produces a subset of the entity combinations as a filtered zip list. This is used e.g. to get all the scans for a single subject.

This `get_filtered_ziplist_index()` function performs `filter_list()` twice:

1. Using the `subj_wildcards` (e.g.: `'subject': '{subject}'`) to get a subject/session-specific `zip_list`.
2. To return the indices from that list of the matching wildcards.

In this example, if the wildcards parameter was:

```
{'dir': 'PA', 'acq': '99', 'subject': '01'}
```

Then the first (subject/session-specific) filtered list provides this zip list:

```
{
    'dir': ['AP','PA','AP','PA'],
    'acq': ['98','98','99','99'],
    'subject': ['01','01','01','01']
}
```

which has 4 combinations, and thus are indexed from 0 to 3.

The returned value would then be the index (or indices) that matches the wildcards. In this case, since the wildcards were `{'dir': 'PA', 'acq': '99', 'subject':'01'}`, the return index is 3.

```
>>> snakebids.get_filtered_ziplist_index(
...     {
...         'dir': ['AP','PA','AP','PA', 'AP','PA','AP','PA'],
...         'acq': ['98','98','98','98','99','99','99','99'],
...         'subject': ['01','01','02','02','01','01','02','02' ]
...     },
...     {'dir': 'PA', 'acq': '99', 'subject': '01'},
...     {'subject': '{subject}' }
... )
3
```

7.7.4 Data Structures

class snakebids.**BidsComponent**(*, *name*, *path*, *zip_lists*)

Representation of a bids data component.

A component is a set of data entries all corresponding to the same type of object. Entries vary over a set of entities. For example, a component may represent all the unprocessed, T1-weighted anatomical images aqcuired from a group of 100 subjects, across 2 sessions, with three runs per session. Here, the subject, session, and run are the entities over which the component varies. Each entry in the component has a single value assigned for each of the three entities (e.g subject 002, session 01, run 1).

Each entry can be defined solely by its wildcard values. The complete collection of entries can thus be stored as a table, where each row represents an entity and each column represents an entry.

`BidsComponent` stores and indexes this table. It uses 'row-first' indexing, meaning first an entity is selected, then an entry. It also has a number of properties and methods making it easier to incorporate the data in a snakemake workflow.

In addition, `BidsComponent` stores a template *~BidsComponent.path* derived from the source dataset. This path is used by the *expand()* method to recreate the original filesystem paths.

The real power of the `BidsComponent`, however, is in creating derived paths based on the original dataset. Using the :meth`~BidsComponent.expand` method, you can pass new paths with `{wildcard}` placeholders wrapped in braces and named according to the entities in the component. These placeholders will be substituted with the entity values saved in the table, giving you a list of paths the same length as the number of entries in the component.

BidsComponents are immutable: their values cannot be altered.

Parameters

- **name** (*str*) –
- **path** (*str*) –

name: `str`

Name of the component

path: `str`

Wildcard-filled path that matches the files for this component.

expand(*paths=None*, */*, *allow_missing=False*, ***wildcards*)

Safely expand over given paths with component wildcards.

Uses the entity-value combinations found in the dataset to expand over the given paths. If no path is provided, expands over the component *path* (thus returning the original files used to create the component). Extra wildcards can be specified as keyword arguments.

By default, expansion over paths with extra wildcards not accounted for by the component causes an error. This prevents accidental partial expansion. To allow the passage of extra wildcards without expansion,set `allow_missing` to `True`.

Uses the snakemake expand under the hood.

Parameters

- **paths** (*Iterable[Path | str] | Path | str | None*) – Path or list of paths to expand over. If not provided, the component's own *path* will be expanded over.
- **allow_missing** (*bool | str | Iterable[str]*) – If True, allow `{wildcards}` in the provided paths that are not present either in the component or in the extra provided `**wildcards`. These wildcards will be preserved in the returned paths.
- **wildcards** (*str | Iterable[str]*) – Each keyword should be the name of an wildcard in the provided paths. Keywords not found in the path will be ignored. Keywords take values or lists of values to be expanded over the provided paths.

Return type
list[str]

property entities: `MultiSelectDict[str, list[str]]`

Component entities and their associated values.

Dictionary where each key is an entity and each value is a list of the unique values found for that entity. These lists might not be the same length.

filter(*, *regex_search=False*, ***filters*)

Filter component based on provided entity filters.

This method allows you to expand over a subset of your wildcards. This could be useful for extracting subjects from a specific patient group, running different rules on different aquisitions, and any other reason you may need to filter your data after the workflow has already started.

Takes entities as keyword arguments assigned to values or list of values to select from the component. Only columns containing the provided entity-values are kept. If no matches are found, a component with the all the original entities but with no values will be returned.

Returns a brand new *BidsComponent*. The original component is not modified.

Parameters

- **regex_search** (*bool | str | Iterable[str]*) – Treat filters as regex patterns when matching with entity-values.
- **filters** (*str | Iterable[str]*) – Each keyword should be the name of an entity in the component. Entities not found in the component will be ignored. Keywords take values or a list of values to be matched with the component *zip_lists*

Return type
Self

pformat(*max_width=None*, *tabstop=4*)

Pretty-format component.

Parameters

- **max_width** (*int | float | None*) – Maximum width of characters for output. If possible, zip_list table will be elided to fit within this width
- **tabstop** (*int*) – Number of spaces for output indentation

Return type
str

property wildcards: MultiSelectDict[str, str]

Wildcards in brace-wrapped syntax.

Dictionary where each key is the name of a wildcard entity, and each value is the Snakemake wildcard used for that entity.

property zip_lists

Table of unique wildcard groupings for each member in the component.

Dictionary where each key is a wildcard entity and each value is a list of the values found for that entity. Each of these lists has length equal to the number of images matched for this modality, so they can be zipped together to get a list of the wildcard values for each file.

Legacy BidsComponents properties

The following properties are historical aliases of BidsComponents properties. There are no current plans to deprecate them, but new code should avoid them.

property BidsComponent.**input_zip_lists:** *snakebids.types.ZipList*

Alias of *zip_lists*.

Dictionary where each key is a wildcard entity and each value is a list of the values found for that entity. Each of these lists has length equal to the number of images matched for this modality, so they can be zipped together to get a list of the wildcard values for each file.

property BidsComponent.**input_wildcards**

Alias of *wildcards*

Wildcards in brace-wrapped syntax.

property BidsComponent.**input_name:** str

Alias of *name*.

Name of the component

property BidsComponent.**input_path:** str

Alias of *path*.

Wildcard-filled path that matches the files for this component.

property BidsComponent.**input_lists**

Alias of *entities*

Component entities and their associated values.

class snakebids.**BidsPartialComponent**(*, *zip_lists*)

Primitive representation of a bids data component.

See *BidsComponent* for an extended definition of a data component.

BidsPartialComponents are typically derived from a *BidsComponent*. They do not store path information, and do not represent real data files. They just have a table of entity-values, typically a subset of those present in their source *BidsComponent*.

Despite this, BidsPartialComponents still allow you to expand the data table over new paths, allowing you to derive paths from your source dataset.

The members of BidsPartialComponent are identical to *BidsComponent* with the following exceptions:

- No *name* or *path*
- expand() must be given a path or list of paths as the first argument

BidsPartialComponents are immutable: their values cannot be altered.

class snakebids.**BidsComponentRow**(*iterable*, /, *entity*)

A single row from a BidsComponent.

This class is derived by indexing a single entity from a *BidsComponent* or *BidsPartialComponent*. It should not be constructed manually.

The class is a subclass of ImmutableList and can thus be treated as a tuple. Indexing it via row[<int>] gives the entity-value of the selected entry.

The *entities* and *wildcards* directly return the list of unique entity-values or the {brace-wrapped-entity} name corresponding to the row, rather than a dict.

The *expand()* and *filter()* methods behave as they would in a *BidsComponent* with a single entity.

Parameters

- **iterable** (*Iterable[str]*) –
- **entity** (*str*) –

property entities: tuple[str, ...]

The unique values associated with the component.

property wildcards: str

The entity name wrapped in wildcard braces.

expand(*paths*, */*, *allow_missing=False*, ***wildcards*)

Safely expand over given paths with component wildcards.

Uses the entity-values represented by this row to expand over the given paths. Extra wildcards can be specified as keyword arguments.

By default, expansion over paths with extra wildcards not accounted for by the component causes an error. This prevents accidental partial expansion. To allow the passage of extra wildcards without expansion,set `allow_missing` to `True`.

Uses the snakemake expand under the hood.

Parameters

- **paths** (*Iterable[Path | str] | Path | str*) – Path or list of paths to expand over
- **allow_missing** (*bool | str | Iterable[str]*) – If True, allow {`wildcards`} in the provided paths that are not present either in the component or in the extra provided `**wildcards`. These wildcards will be preserved in the returned paths.
- **wildcards** (*str | Iterable[str]*) – Each keyword should be the name of an wildcard in the provided paths. Keywords not found in the path will be ignored. Keywords take values or lists of values to be expanded over the provided paths.

Return type
list[str]

filter(*spec=None*, */*, ***, *regex_search=False*, ***filters*)

Filter component based on provided entity filters.

Extracts a subset of the entity-values present in the row.

Takes entities as keyword arguments assigned to values or list of values to select from the component. Only columns containing the provided entity-values are kept. If no matches are found, a component with the all the original entities but with no values will be returned.

Returns a brand new *BidsComponentRow*. The original component is not modified.

Parameters

- **spec** (*Iterable[str] | str | None*) – Value or iterable of values assocatiated with the ComponentRow's `entity`. Equivalent to specifying `.filter(entity=value)`
- **regex_search** (*bool | str | Iterable[str]*) – Treat filters as regex patterns when matching with entity-values.
- **filters** (*str | Iterable[str]*) – Keyword-value(s) filters as in *filter()*. Here, the only valid filter is the `entity` of the *BidsComponentRow*; all others will be ignored.

Return type
Self

class snakebids.**BidsDataset**(*data*, *layout=None*)

A bids dataset parsed by pybids, organized into BidsComponents.

BidsDatasets are typically generated using *generate_inputs()*, which reads the `pybids_inputs` field in your snakemake config file and, for each entry, creates a BidsComponent using the provided name, wildcards, and filters.

Individual components can be accessed using bracket-syntax: (e.g. `inputs["t1w"]`).

Provides access to summarizing information, for instance, the set of all subjects or sessions found in the dataset.

Parameters

- **data** (*Any*) –
- **layout** (*BIDSLayout | None*) –

layout: BIDSLayout | None

Underlying layout generated from pybids. Note that this will be set to None if custom paths are used to generate every *component*

pformat(*max_width=None*, *tabstop=4*)

Pretty-format dataset.

Parameters

- **max_width** (*int | float | None*) – Maximum width of characters for output. If possible, zip_list table will be elided to fit within this width
- **tabstop** (*int*) – Number of spaces for output indentation

Return type
str

property path: dict[str, str]

Dict mapping *BidsComponent* names to their `paths`.

> **Warning:** Deprecated since version 0.8.0: The behaviour of path will change in an upcoming release, where it will refer instead to the root path of the dataset. Please access component paths using *Dataset[<component_name>].path*

property zip_lists: dict[str, *snakebids.types.ZipList*]

Dict mapping *BidsComponent* names to their `zip_lists`.

> **Warning:** Deprecated since version 0.8.0: The behaviour of zip_lists will change in an upcoming release, where it will refer instead to the consensus of entity groups across all components in the dataset. Please access component zip_lists using *Dataset[<component_name>].zip_lists*

property entities: dict[str, snakebids.utils.containers.MultiSelectDict[str, list[str]]]

Dict mapping *BidsComponent* names to their *entities*.

> **Warning:** Deprecated since version 0.8.0: The behaviour of entities will change in the 1.0 release, where it will refer instead to the union of all entity-values across all components in the dataset. Please access component entity lists using *Dataset[<component_name>].entities*

property wildcards: dict[str, snakebids.utils.containers.MultiSelectDict[str, str]]

Dict mapping *BidsComponent* names to their *wildcards*.

> **Warning:** Deprecated since version 0.8.0: The behaviour of wildcards will change in an upcoming release, where it will refer instead to the union of all entity-wildcard mappings across all components in the dataset. Please access component wildcards using *Dataset[<component_name>].wildcards*

property subjects: list[str]

A list of the subjects in the dataset.

property sessions: list[str]

A list of the sessions in the dataset.

property subj_wildcards: dict[str, str]

The subject and session wildcards applicable to this dataset.

`{"subject":"{subject}"}` if there is only one session, `{"subject": "{subject}", "session": "{session}"}` if there are multiple sessions.

property as_dict: *BidsDatasetDict*

Get the layout as a legacy dict.

Included primarily for backward compatability with older versions of snakebids, where generate_inputs() returned a dict rather than the *BidsDataset* class

Return type
BidsDatasetDict

classmethod from_iterable(*iterable*, *layout=None*)

Construct Dataset from iterable of BidsComponents.

Parameters

- **iterable** (*Iterable[BidsComponent]*) –
- **layout** (*BIDSLayout | None*) –

Return type
BidsDataset

Legacy BidsDataset properties

The following properties are historical aliases of *BidsDataset* properties. There are no current plans to deprecate them, but new code should avoid them.

property BidsDataset.**input_zip_lists:** dict[str, snakebids.utils.containers.MultiSelectDict[str, list[str]]]

Alias of *zip_lists*

Dict mapping *BidsComponent* names to their zip_lists.

property BidsDataset.**input_wildcards**: dict[str, snakebids.utils.containers.MultiSelectDict[str, str]]

Alias of *wildcards*

Dict mapping *BidsComponent* names to their *wildcards*.

property BidsDataset.**input_path**: dict[str, str]

Alias of *path*

Dict mapping *BidsComponent* names to their `paths`.

property BidsDataset.**input_lists**: dict[str, snakebids.utils.containers.MultiSelectDict[str, list[str]]]

Alias of *entities*

Dict mapping *BidsComponent* names to their *entities*.

class snakebids.**BidsDatasetDict**

Dict equivalent of BidsInputs, for backwards-compatibility.

7.7.5 BIDS App Bootstrapping

Tools to generate a Snakemake-based BIDS app.

This legacy module once had the core snakebids bidsapp implementation, but now is a simple wrapper around *snakebids.bidsapp* with the *SnakemakeBidsApp* plugin. For new apps, this functionality can be more flexibly implemented with:

```
from snakebids import bidsapp, plugins

bidsapp.app([plugins.SnakemakeBidsApp(...)])
```

class snakebids.app.**SnakeBidsApp**(*snakemake_dir*, *plugins=_Nothing.NOTHING*, *skip_parse_args=False*, *parser=None*, *configfile_path=None*, *snakefile_path=None*, *config=None*, *version=None*, *args=None*)

Snakebids app with config and arguments.

Parameters

- **snakemake_dir** (*str | Path*) – Root directory of the snakebids app, containing the config file and workflow files.
- **plugins** (*list[Callable[[snakebids.app.SnakeBidsApp], None | snakebids.app.SnakeBidsApp]]*) – List of plugins to be registered.

  See *Using plugins* for more info.
- **skip_parse_args** (*bool*) – DEPRECATED: no-op.
- **parser** (*Any*) – DEPRECATED: no-op. (Historic: Parser including only the arguments specific to this Snakebids app, as specified in the config file. By default, it will use *create_parser()* from *cli.py*)
- **configfile_path** (*pathlib.Path | None*) – Relative path to config file (relative to snakemake_dir). By default, autocalculates based on snamake_dir
- **snakefile_path** (*pathlib.Path | None*) – Absolute path to the input Snakefile. By default, autocalculates based on snakemake_dir:

```
join(snakemake_dir, snakefile_path)
```

- **config** (*Any*) – DEPRECATED: no-op. (Historic: Contains all the configuration variables parsed from the config file and generated during the initialization of the SnakeBidsApp.)
- **args** (*Any*) – DEPRECATED: no-op. (Historic: Arguments to use when running the app. By default, generated using the parser attribute, autopopulated with args from *config.py*)
- **version** (*str | None*) – DEPRECATED: no-op, use version plugin instead

property config

Get config dict (before arguments are parsed).

property parser

Get parser.

run_snakemake()

Run snakemake with the given config, after applying plugins.

Return type
None

create_descriptor(*out_file*)

Generate a boutiques descriptor for this Snakebids app.

Parameters
out_file (*PathLike[str] | str*) –

Return type
None

BIDS App

snakebids.bidsapp.**app**(*plugins=None*, *, *config=None*, ***argparse_args*)

Create a BIDSApp.

Parameters

- **plugins** (*Iterable[_Plugin] | None*) – List of snakebids plugins to apply to app (see *Using plugins*).
- **config** (*dict[str, Any] | None*) – Initial config. Will be updated with parsed arguments from `parser` and potentially modified by plugins
- ****argparse_args** (*Unpack[ArgumentParserArgs]*) – Arguments passed on transparently to `argparse.ArgumentParser`.

Return type
_Runner

class snakebids.bidsapp.run.**_Runner**

Runtime manager for BIDS apps.

Manages the parser, config, and plugin relay for snakebids BIDS apps. This class should not be constructed directly, but built using the *bidsapp.app()* function.

Parameters

- **pm** (*pluggy.PluginManager*) –
- **parser** (*argparse.ArgumentParser*) –

- **argument_groups** (*ArgumentGroups*) –

pm: pluggy.PluginManager

Reference to the plugin manager.

parser: argparse.ArgumentParser

Parser used for parsing CLI arguments.

The parser may be manipulated before running action methods to add initial arguments (via `parser.add_argument()`) and argument groups (via `parser.add_argument_group()`). Any argument groups added will automatically be indexed in *argument_groups* and made available to plugins via their `title`.

config: dict[str, Any]

Configuration dictionary for passing data between plugins.

argument_groups: ArgumentGroups

Argument group reference accessible to plugins for organizing CLI arguments.

Any groups added to the *parser* before the action methods are called will be automatically added, indexed by their titles. Thus, this object should typically only be used within plugins.

Action Methods

Plugins are only run when calling the action methods. These methods trigger running of the plugins up to a specified point. For example, *build_parser()* runs the `initialize_config()` and `add_cli_arguments()` hooks. It can thus be used to build the parser without actually parsing any arguments.

Plugins are always run in the same order. Action methods will always trigger the entire plugin chain up until their stopping point. Importantly, plugins will only ever be run once, even if action methods are called multiple times. For example, if *build_parser()* is called, then *parse_args()*, the parser will only be built once.

build_parser()

Run plugins affecting the `parser` without yet parsing arguments.

parse_args(*args=None*)

Run all plugins and parse arguments.

Parameters
args (*list[str] | None*) –

run(*args=None*)

Parameters
args (*list[str] | None*) –

7.7.6 Path Building

bids	Generate bids or bids-like paths
bids_factory	Create new *bids()* functions according to a spec

7.7.7 Dataset Creation

generate_inputs	Dynamically generate snakemake inputs using pybids_inputs.

7.7.8 Dataset Manipulation

filter_list	Filter zip_list, including only entries with provided entity values.
get_filtered_ziplist_index	Return the indices of all entries matching the filter query.

7.7.9 Data Structures

BidsComponent	Representation of a bids data component.
BidsPartialComponent	Primitive representation of a bids data component.
BidsComponentRow	A single row from a BidsComponent.
BidsDataset	A bids dataset parsed by pybids, organized into BidsComponents.
BidsDatasetDict	Dict equivalent of BidsInputs, for backwards-compatibility.

7.7.10 BIDS App Booststrapping

SnakeBidsApp	Snakebids app with config and arguments.

7.8 Internals

Note: These types are mostly used internally. The API most users will need is documented in *the main API page*, but these are listed here for reference (and some of the main API items link here).

7.8.1 utils

class snakebids.utils.utils.**BidsTag**

Interface for BidsTag configuration.

snakebids.utils.utils.**DEPRECATION_FLAG = '<!DEPRECATED!>'**

Sentinel string to mark deprecated config features

snakebids.utils.utils.**read_bids_tags**(*bids_json=None*)

Read the bids tags we are aware of from a JSON file.

This is used specifically for compatibility with pybids, since some tag keys are different from how they appear in the file name, e.g. `subject` for `sub`, and `acquisition` for `acq`.

Parameters
bids_json (*Path | None*) – Path to JSON file to use, if not specified will use `bids_tags.json` in the snakebids module.

Returns
Dictionary of bids tags

Return type
dict

class snakebids.utils.utils.**BidsEntity**(*entity*)

Bids entities with tag and wildcard representations.

Parameters
entity (*str*) –

property tag: str

Get the bids tag version of the entity.

For entities in the bids spec, the tag is the short version of the entity name. Otherwise, the tag is equal to the entity.

property match: str

Get regex of acceptable value matches.

If no pattern is associated with the entity, the default pattern is a word with letters and numbers

property before: str

Regex str to search before value in paths.

property after: str

Regex str to search after value in paths.

property regex: Pattern[str]

Complete pattern to match when searching in paths.

Contains three capture groups, the first corresponding to "before", the second to "value", and the third to "after"

property wildcard: str

Get the snakebids {wildcard}.

The wildcard is generally equal to the tag, i.e. the short version of the entity name, except for subject and session, which use the full name name. This is to ensure compatibility with the bids function

classmethod from_tag(*tag*)

Return the entity associated with the given tag, if found.

If not associated entity is found, the tag itself is used as the entity name

Parameters
tag (*str*) – tag to search

Return type
BidsEntity

classmethod normalize(*item*, */*)

Return the entity associated with the given item, if found.

Supports both strings and BidsEntities as input. Unlike the constructor, if a tag name is given, the associated entity will be returned. If no associated entity is found, the tag itself is used as the entity name

Parameters
item (*str | BidsEntity*) – tag to search

Return type
BidsEntity

snakebids.utils.utils.**matches_any**(*item*, *match_list*, *match_func*, **args*)

Test if item matches any of the items in match_list.

Parameters

- **item** (*_T*) – Item to test
- **match_list** (*Iterable[_T]*) – Items to compare with
- **match_func** (*BinaryOperator[_T, object]*) – Function to test equality. Defaults to basic equality (==) check
- **args** (*Any*) –

Return type
bool

exception snakebids.utils.utils.**BidsParseError**(*path*, *entity*)

Exception raised for errors encountered in the parsing of Bids paths.

Parameters

- **path** (*str*) –
- **entity** (*BidsEntity*) –

Return type
None

snakebids.utils.utils.**property_alias**(*prop*, *label=None*, *ref=None*, *copy_extended_docstring=False*)

Set property as an alias for another property.

Copies the docstring from the aliased property to the alias

Parameters

- **prop** (*property*) – Property to alias
- **label** (*str | None*) – Text to use in link to aliased property
- **ref** (*str | None*) – Name of the property to alias
- **copy_extended_docstring** (*bool*) – If True, copies over the entire docstring, in addition to the summary line

Return type
property

snakebids.utils.utils.**surround**(*s*, *object_*, */*)

Surround a string or each string in an iterable with characters.

Parameters

- **s** (*Iterable[str] | str*) –

- **object_** (*str*) –

Return type
Iterable[str]

snakebids.utils.utils.**zip_list_eq**(*first*, *second*, */*)

Compare two zip lists, allowing the order of columns to be irrelevant.

Parameters

- **first** (*Mapping[str, Sequence[str]]*) –
- **second** (*Mapping[str, Sequence[str]]*) –

snakebids.utils.utils.**get_first_dir**(*path*)

Return the top level directory in a path.

If absolute, return the root. This function is necessary to handle paths with `./`, as `pathlib.Path` filters this out.

Parameters
path (*str*) –

Return type
str

snakebids.utils.utils.**to_resolved_path**(*path*)

Convert provided object into resolved path.

Parameters
path (*str | PathLike[str]*) –

snakebids.utils.utils.**get_wildcard_dict**(*entities*, */*)

Turn entity strings into wildcard dicts as {"entity": "{entity}"}.

Parameters
entities (*str | Iterable[str]*) –

Return type
dict[str, str]

snakebids.utils.utils.**entity_to_wildcard**(*entities*, */*)

Turn entity strings into wildcard dicts as {"entity": "{entity}"}.

Parameters
entities (*str | Iterable[str]*) –

snakebids.utils.utils.**text_fold**(*text*)

Fold a block of text into a single line as in yaml folded multiline string.

Parameters
text (*str*) –

7.8.2 snakemake_io

File globbing functions based on snakemake.io library.

snakebids.utils.snakemake_io.**regex**(*filepattern*)

Build Snakebids regex based on the given file pattern.

Parameters
filepattern (*str*) –

Return type
str

snakebids.utils.snakemake_io.**glob_wildcards**(*pattern*, *files=None*, *followlinks=False*)

Glob the values of wildcards by matching a pattern to the filesystem.

Returns a zip_list of field names with matched wildcard values.

Parameters

- **pattern** (*str | Path*) – Path including wildcards to glob on the filesystem.
- **files** (*Sequence[str | Path] | None*) – Files from which to glob wildcards. If None (default), the directory corresponding to the first wildcard in the pattern is walked, and wildcards are globbed from all files.
- **followlinks** (*bool*) – Whether to follow links when globbing wildcards.

Return type
snakebids.types.ZipList

snakebids.utils.snakemake_io.**update_wildcard_constraints**(*pattern*, *wildcard_constraints*, *global_wildcard_constraints*)

Update wildcard constraints.

Parameters

- **pattern** (*str*) – Pattern on which to update constraints.
- **wildcard_constraints** (*dict*) – Dictionary of wildcard:constraint key-value pairs.
- **global_wildcard_constraints** (*dict*) – Dictionary of wildcard:constraint key-value pairs.

Return type
str

7.8.3 exceptions

exception snakebids.exceptions.**ConfigError**(*msg*)

Exception raised for errors with the Snakebids config.

Parameters
msg (*str*) –

Return type
None

exception snakebids.exceptions.**RunError**(*msg*, **args*)

Exception raised for errors in generating and running the snakemake workflow.

Parameters

- **msg** (*str*) –
- **args** (*object*) –

Return type
None

exception snakebids.exceptions.**PybidsError**

Exception raised when pybids encounters a problem.

exception snakebids.exceptions.**DuplicateComponentError**(*duplicated_names*)

Raised when a dataset is constructed from components with the same name.

Parameters
duplicated_names (*Iterable[str]*) –

exception snakebids.exceptions.**MisspecifiedCliFilterError**(*misspecified_filter*)

Raised when a magic CLI filter cannot be parsed.

Parameters
misspecified_filter (*str*) –

exception snakebids.exceptions.**SnakebidsPluginError**

Exception raised when a Snakebids plugin encounters a problem.

7.8.4 types

class snakebids.types.**FilterSpec**

Optional filter specification allowing regex matching.

class snakebids.types.**InputConfig**

Configuration for a single bids component.

filters: `Mapping[str, str | bool | Sequence[str | bool] | FilterSpec]`

Filters to pass on to `BIDSLayout.get()`

Each key refers to the name of an entity. Values may take the following forms:

- `string`: Restricts the entity to the exact string given
- `bool`: `True` requires the entity to be present (with any value). `False` requires the entity to be absent.
- `list` [`str`]: List of allowable values the entity may take.

In addition, a few special filters may be added which carry different meanings:

- `use_regex: True`: If present, all strings will be interpreted as regex
- **scope: Restricts the scope of the component. It may take the following values:**
  - `"all"`: search everything (default behaviour)
  - `"raw"`: only search the top-level raw dataset
  - `"derivatives"`: only search derivative datasets
  - **<PipelineName>: only search derivative datasets with a matching pipeline** name

wildcards: list[str]

Wildcards to allow in the component.

Each value in the list refers to the name of an entity. If the entity is present, the generated *BidsComponent* will have values of this entity substituted for wildcards in the *path*, and the entity will be included in the *zip_lists*.

If the entity is not found, it will be ignored.

class snakebids.types.**BinaryOperator**(**args*, ***kwargs*)

Callables that act on two objects of identical type.

class snakebids.types.**Expandable**(**args*, ***kwargs*)

Protocol represents objects that hold an entity table and can expand over a path.

Includes BidsComponent, BidsPartialComponent, and BidsComponentRow

class snakebids.types.**MultiSelectable**(**args*, ***kwargs*)

Mappings supporting selection with multiple keys.

class snakebids.types.**OptionalFilterType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Sentinel value for CLI OPTIONAL filtering.

This is necessary because None means no CLI filter was added.

snakebids.types.**InputsConfig**

Configuration for all bids components to be parsed in the app

Should be defined in the config.yaml file, by convention in a key called 'pybids_inputs'

alias of Dict[str, *InputConfig*]

snakebids.types.**ZipList**

Multiselectable dict mapping entity names to possible values.

All lists must be the same length. Entries in each list with the same index correspond to the same path. Thus, the ZipList can be read like a table, where each row corresponds to an entity, and each "column" corresponds to a path.

alias of MultiSelectDict[str, List[str]]

snakebids.types.**ZipListLike**

Generic form of a *ZipList*

Useful for typing functions that won't mutate the ZipList or use MultiSelectDict capabilities. Like *ZipList*, each Sequence must be the same length, and values in each with the same index must correspond to the same path.

alias of Mapping[str, Sequence[str]]

class snakebids.types.**ZipList**

class snakebids.types.**InputsConfig**

class snakebids.types.**ZipListLike**

7.9 Plugins

Note: This page consists of plugins distributed with Snakebids. Externally developed and distributed plugins may be missing from this page.

7.9.1 Core

These plugins provide essential bids app features and will be needed by most apps.

class snakebids.plugins.**BidsArgs**(*argument_group=''*, *bids_dir=True*, *output_dir=True*, *analysis_level=True*, *analysis_level_choices=None*, *analysis_level_choices_config=None*, *participant_label=True*, *exclude_participant_label=True*, *derivatives=True*)

Add basic BIDSApp arguments.

Parameters

- **argument_group** (*str*) – Specify title of the group to which arguments should be added
- **bids_dir** (*bool*) – Indicate if bids_dir (first bids argument) should be defined
- **output_dir** (*bool*) – Indicate if output_dir (second bids argument) should be defined
- **analysis_level** (*bool*) – Indicate if output_dir (third bids argument) should be defined
- **analysis_level_choices** (*list[str] | None*) – List of valid analysis levels
- **analysis_level_choices_config** (*str | None*) – Configuration key containing analysis level choices
- **participant_label** (*bool*) – Indicate if `participant_label` should be defined. Used to filter specific subjects for processesing
- **exclude_participant_label** (*bool*) – Indicate if `exclude_participant_label` should be defined. Used to excluded specific subjects from processesing
- **derivatives** (*bool*) – Indicate if `derivatives` should be defined. Used to allow automatic derivative indexing or specify paths to derivatives.

CLI Arguments

All arguments are added by default, but can be disabled using their respective parameters. Additinally, arguments can be directly overriden by adding arguments to the parser before the plugin runs, using the following `dests`:

- `bids_dir`: The input bids directory
- `output_dir`: The output bids directory
- `analysis_level`: The level of analysis to perform (usually `participant` or `group`)
- `participant_label`: Collection of subject labels to include in analysis
- `exclude_participant_label`: Collection of subject labels to exclude from analysis
- `derivatives`: Collection of derivative folder paths to include in bids indexing.

Note that only the `dest` field matters when overriding arguments, all other feature, including positional versus optional, are incidental. Other than adding arguments for the above `dests`, `BidsArgs` makes no assumptions about these arguments. Other plugins, however, may expect to find data in the parser namespace or config corresponding to the `dest` names.

Warning: Overriding just one or two of the positional arguments may alter the order, preventing the app from being called correctly. Thus, if any of the positional args are being overriden, they all should be.

Analysis levels

Valid analysis levels can be set using the `analysis_level_choices` key or the `analysis_level_choices_config` key. If both are set, an error will be raised. If neither are set, the config will first be searched for the `analysis_levels` key. If this is not found, analysis levels will be set to `["participant", "group"]`.

add_cli_arguments(*parser*, *argument_groups*, *config*)

Add arguments from config.

Parameters

- **parser** (*argparse.ArgumentParser*) –
- **argument_groups** (*ArgumentGroups*) –
- **config** (*dict[str, Any]*) –

class snakebids.plugins.**ComponentEdit**(*components_key='pybids_inputs'*)

Use CLI arguments to edit the filters, wildcards, and paths of components.

Arguments are added based on the components specified in `config`. For each component, a `--filter-<comp_name>`, `--wildcards-<comp_name>`, and a `--path-<comp_name>` argument will be added to the CLI. After parsing, these arguments are read and used to update the original component specification within config.

CLI arguments created by this plugin cannot be overriden.

Parameters

components_key (*str*) – Key of component specification within the config dictionary.

add_cli_arguments(*parser*, *config*)

Add filter, wildcard, and path override arguments for each component.

Parameters

- **parser** (*ArgumentParser*) –
- **config** (*dict[str, Any]*) –

update_cli_namespace(*namespace*, *config*)

Apply provided overrides to the component configuration.

Parameters

- **namespace** (*dict[str, Any]*) –
- **config** (*dict[str, Any]*) –

class snakebids.plugins.**CliConfig**(*cli_key='parse_args'*)

Configure CLI arguments directly in config.

Arguments are provided in config in a dictionary stored under `cli_key`. Each entry maps the name of the argument to a set of valid arguments for `add_argument()`.

This plugin will attempt to be the first to add arguments, and thus can be used to override arguments from other compatible plugins, such as *BidsArgs*

Parameters

cli_key (*str*) – Key of dict containing arguments

Example

```
parse_args:
    --tunable-parameter:
        help: |
            A parameter important to the analysis that you can be set from the
            commandline. If not set, a sensible default will be used
        default: 5
        type: float
    --alternate-mode:
        help: |
            A flag activating a secondary feature of the workflow
        action: store_true
    --derivatives:
        help: |
            An alternate help message for --derivatives, which will override
            the help message from argument given by ``BidsArgs``. Note that we
            must again specify the ``nargs`` and ``type`` for the param
        type: Path
        nargs: "*"
```

add_cli_arguments(*parser*, *config*)

Add arguments from config.

Parameters

- **parser** (*ArgumentParser*) –
- **config** (*dict[str, Any]*) –

class snakebids.plugins.**Pybidsdb**(*argument_group=None*)

Add CLI parameters to specify and reset a pybids database.

Parameters

argument_group (*str | None*) – Specify title of the group to which arguments should be added

CLI Arguments

Two arguments are added to the CLI. These can be overriden by adding arguments with corresponding `dests` before this plugin is run:

- `plugins.pybidsdb.dir`: (`Path`) Path of the database
- `plugins.pybidsdb.reset`: (`bool`) Boolean indicating the database should be reset.

After parsing, the above dests will be moved into `config` under the following names:

- `plugins.pybidsdb.dir` → `pybidsdb_dir`
- `plugins.pybidsdb.reset` → `pybidsdb_reset`

This plugin only handles the CLI arguments, it does not do any actions with the database. The above config entries can be consumed by downstream processes.

add_cli_arguments(*parser*, *argument_groups*)

Add database parameters.

Parameters

- **parser** (*argparse.ArgumentParser*) –
- **argument_groups** (*ArgumentGroups*) –

update_cli_namespace(*namespace*, *config*)

Assign database parameters to config.

Parameters

- **namespace** (*dict[str, Any]*) –
- **config** (*dict[str, Any]*) –

7.9.2 Utility

These plugins add additional features that may be helpful to most bids apps.

class snakebids.plugins.**BidsValidator**(*raise_invalid_bids=True*)

Perform validation of a BIDS dataset using the bids-validator.

If the dataset is not valid according to the BIDS specifications, an InvalidBidsError is raised.

Parameters

raise_invalid_bids (*bool*) – Flag to indicate whether InvalidBidsError should be raised if BIDS validation fails. Default to True.

finalize_config(*config*)

Perform BIDS validation of dataset.

Raises

InvalidBidsError – Raised when the input BIDS directory does not pass validation with the bids-validator

Parameters

config (*dict[str, Any]*) –

Return type

None

7.9.3 Workflow Integrations

These plugins enable integrations with specific workflow managers.

class snakebids.plugins.**SnakemakeBidsApp**(*snakemake_dir*, *, *configfile_path=_Nothing.NOTHING*, *configfile_outpath=None*, *snakefile_path=_Nothing.NOTHING*)

Snakebids app with config and arguments.

Loads the *BidsArgs*, *CliConfig*, *Pybidsdb*, and *ComponentEdit* plugins as dependencies.

Parameters

- **snakemake_dir** (*str | Pathlike[str]*) – Root directory of the snakebids app, containing the config file and workflow files.
- **configfile_path** (*pathlib.Path | None*) – Path of config file. By default, looks for `config.yaml`, `config.json`, `snakebids.yaml`, or `snakebids.json` either within the `snakemake_dir` or within `snakemake_dir/config`
- **configfile_outpath** (*pathlib.Path | None*) – Path of output configfile that will be consumed by the snakemake workflow. This is only necessary if the input configfile is not within the snakemake directory. If given, it should be the same as the path specified under `configfile: ...` within the ``Snakefile`
- **snakefile_path** (*pathlib.Path*) – Absolute path to the input Snakefile. By default, looks for `Snakefile` within the `snakemake_dir` or `snakemake_dir/workflow`

cwd: *Path*

The current working directory for running snakemake.

This is where the `.snakemake` metadata folder will be placed.

force_output: *bool*

Allow specifying outputs in unrecognized, non-empty directories.

After the first run, a `.snakebids` file will be created in the directory to mark the directory as safe for future runs.

classmethod create_empty()

Create empty instance of plugin.

initialize_config(*config*)

Read config file and load into config dict.

Parameters
config (*dict[str, Any]*) –

add_cli_arguments(*parser*)

Add snakemake help and force_output arguments.

Parameters
parser (*ArgumentParser*) –

handle_unknown_args(*args*, *config*)

Add snakemake args to config.

Parameters

- **args** (*list[str]*) –
- **config** (*dict[str, Any]*) –

update_cli_namespace(*namespace*, *config*)

Resolve Paths within the namespace and set target.

Parameters

- **namespace** (*dict[str, Any]*) –
- **config** (*dict[str, Any]*) –

finalize_config(*config*)

Perform final steps for snakemake workflows.

Expects to find `output_dir` in config as a fully resolved `Path`

- Modify `output_dir` to *output/results* if output is set to the app dir
- Set `self.cwd`
- Write config file into output dir
- Add snakemake specific parameters to config

Parameters

config (*dict[str, Any]*) –

run(*config*)

Run snakemake with the given config, after applying plugins.

Parameters

config (*dict[str, Any]*) –

7.9.4 Plugin Development

snakebids.bidsapp.**hookimpl**()

Marker to be imported and used in plugins (and for own implementations).

See `pluggy.HookimplMarker()` for more information. Its parameters are represented here fore reference.

Parameters

- **optionalhook** – If `True`, a missing matching hook specification will not result in an error (by default it is an error if no matching spec is found). See Optional validation.
- **tryfirst** – If `True`, this hook implementation will run as early as possible in the chain of N hook implementations for a specification. See Call time order.
- **trylast** – If `True`, this hook implementation will run as late as possible in the chain of N hook implementations for a specification. See Call time order.
- **wrapper** – If `True` ("new-style hook wrapper"), the hook implementation needs to execute exactly one `yield`. The code before the `yield` is run early before any non-hook-wrapper function is run. The code after the `yield` is run after all non-hook-wrapper functions have run. The `yield` receives the result value of the inner calls, or raises the exception of inner calls (including earlier hook wrapper calls). The return value of the function becomes the return value of the hook, and a raised exception becomes the exception of the hook. See Wrappers.
- **hookwrapper** – If `True` ("old-style hook wrapper"), the hook implementation needs to execute exactly one `yield`. The code before the `yield` is run early before any non-hook-wrapper function is run. The code after the `yield` is run after all non-hook-wrapper function have run

The `yield` receives a `pluggy.Result` object representing the exception or result outcome of the inner calls (including earlier hook wrapper calls). This option is mutually exclusive with `wrapper`. See Old-style wrappers.

- **specname** – If provided, the given name will be used instead of the function name when matching this hook implementation to a hook specification during registration. See Hook-spec name matching.

Specs

The spec contains the hooks recognized by snakebids. Each spec has a specific function name and a set of available arguments, and will be called at a specific time during app initialization.

snakebids.bidsapp.hookspecs.**initialize_config**(*config*)

Modify config dict before creation of CLI parser.

Config should be modified in place.

Parameters
config (*dict[str, Any]*) – Possibly empty dictionary of configuration values

snakebids.bidsapp.hookspecs.**add_cli_arguments**(*parser*, *config*, *argument_groups*)

Add any number of arguments to the argument parser.

If special intervention is not made in the *update_cli_namespace()* hook, argument data will be automatically merged into `config` following argument parsing under the argument *dest*. To avoid naming conflicts, plugins should explicitly set the *dest* of each argument they add to `plugins.<plugin_name>.<argument_name>`.

Parameters

- **parser** (*argparse.ArgumentParser*) – BIDS app CLI parser
- **config** (*dict[str, Any]*) – Configuration dictionary preloaded with values from *initialize_config()*
- **argument_groups** (*ArgumentGroups*) –

snakebids.bidsapp.hookspecs.**get_argv**(*argv*, *config*)

Set or modify the CLI parameters that will be parsed by the parser.

Parameters

- **argv** (*list[str]*) –
- **config** (*dict[str, Any]*) –

Return type
list[str] | None

snakebids.bidsapp.hookspecs.**handle_unknown_args**(*args*, *config*)

If `parse_known_args` enabled, handle unknown arguments.

Parameters

- **args** (*list[str]*) – List of unknown arguments parsed by the argparse parser
- **config** (*dict[str, Any]*) – Configuration dictionary

snakebids.bidsapp.hookspecs.**update_cli_namespace**(*namespace*, *config*)

Interact with argument parsing results before they are merged into `config`.

The `namespace` contains the results from `parse_args()` (equivalent to `vars(Namespace)`). Any modifications made to this `dict` will be carried forward in app initialization. For instance, if an entry is deleleted from `namespace`, it will not be available to downstream plugins or be copied into `config`.

Parameters

- **namespace** (*dict[str, Any]*) – the dictionary of values derived from the argparse namespace
- **config** (*dict[str, Any]*) – Configuration dictionary

snakebids.bidsapp.hookspecs.**finalize_config**(*config*)

Perform modifications to the config following merging of parsed arguments.

Parameters
config (*dict[str, Any]*) – Configuration dictionary

snakebids.bidsapp.hookspecs.**run**(*config*)

Consume configuration and perform actions.

This hook runs directly after *finalize_config()*. The config should not be modified.

Parameters
config (*dict[str, Any]*) – The finalized configuration dictionary

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z