DataCurator Documentation

October 30, 2022

Contents

Part I

Index

Chapter 1

DataCurator.jl Documentation

A multithreaded package to validate, curate, and transform large heterogeneous datasets using reproducible recipes, which can be created both in TOML human readable format, or in Julia.

DataCurator is a Swiss army knife that ensures:

- pipelines can focus on the algorithm/problem solving
- human readable `recipes` for future reproducibility
- validation huge datasets at high speed
- out-of-the-box operation without the need for code or dependencies

1.1 Quickstart

We'll show 2 simple examples on how to get started, for a more complete manual please see individual sections in the left pane.

Validate

Check that a directory only contains CSV files, list them in a file, and list any file that's incorrect.

```
[global]
inputdirectory = "testdir"
[any]
conditions=["is_csv_file"]
actions = [["log_to_file", "non_csvs.txt"]]
counter_actions=[["log_to_file", "csvs.txt"]]
```

Execute:

```
./DataCurator.sif -r myrecipe.toml
```

Curate

Flatten all **.txt** files, `flatten` refers to extracting all files from a nested hierarchy (a directory with many subdirectories, each with their own subdirectories and so forth) into 1 single set of files in 1 directory, for ease of processing.

Create a `recipe.toml` file with:

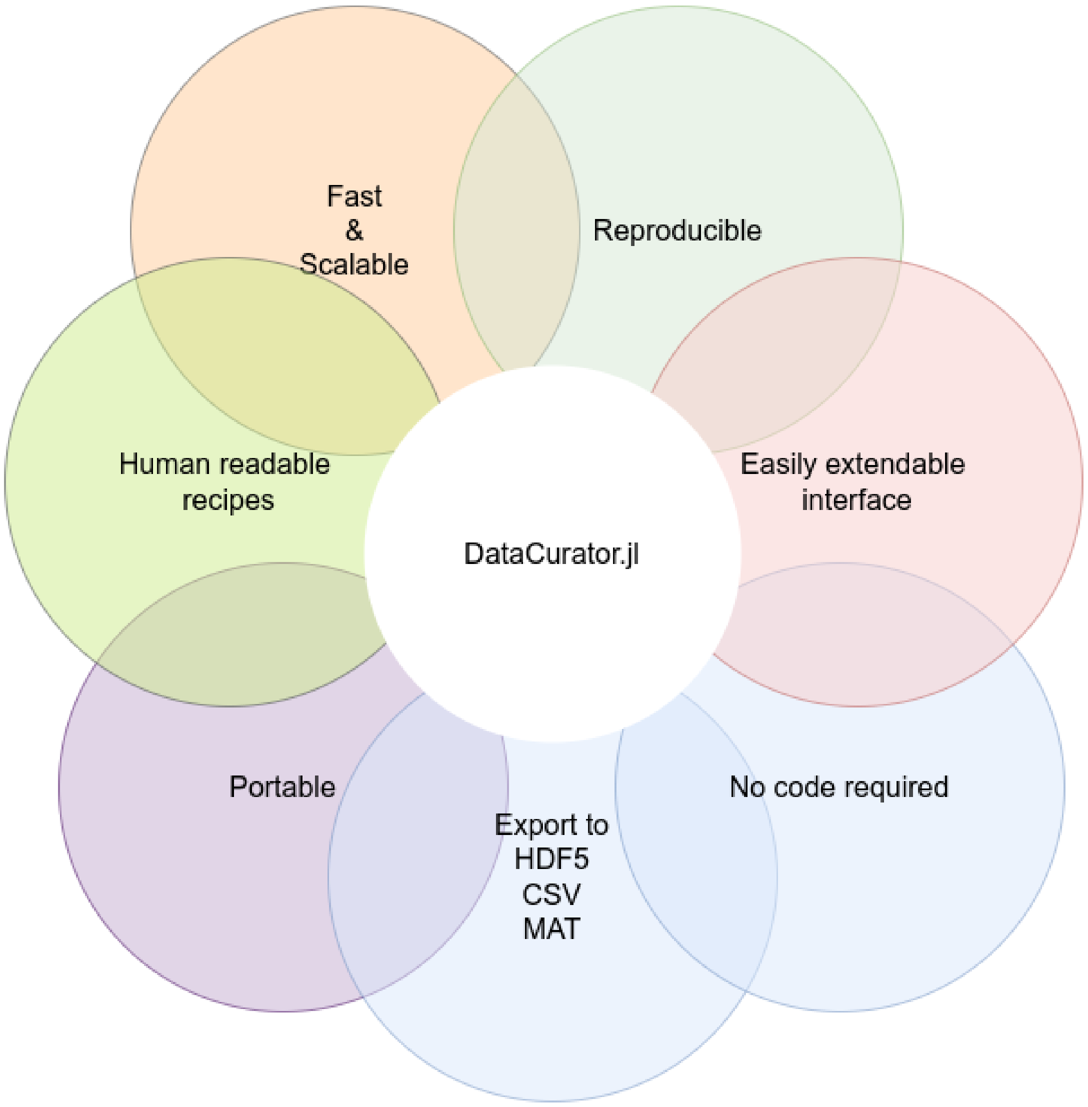


Figure 1.1: Concept

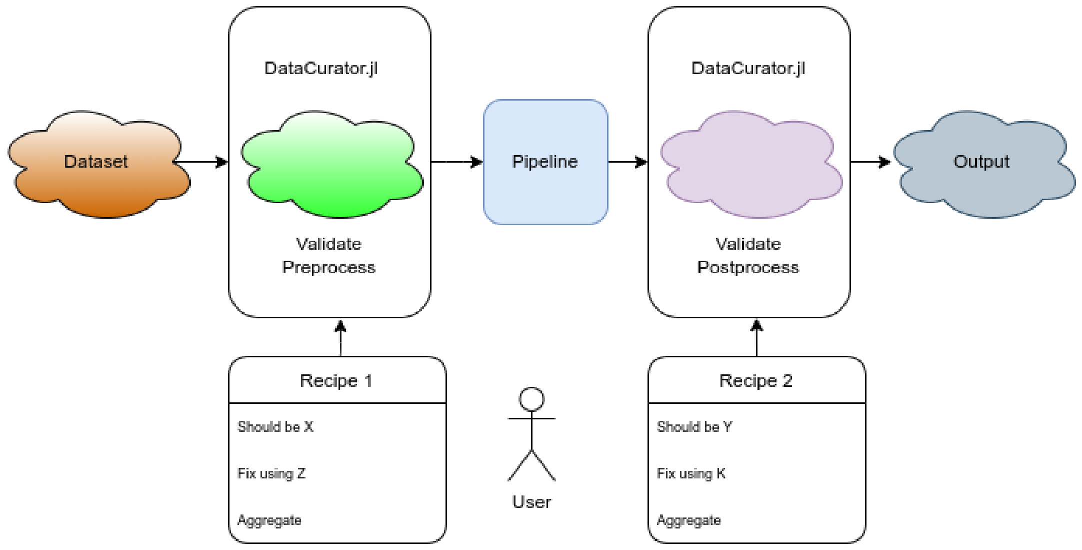


Figure 1.2: Concept

```
[global]
inputdirectory = "testdir"
regex=true
[any]
all=true
conditions = ["isfile", ["endswith", ".*.txt"]]
actions = [["flatten_to", "outdir"]]
```

```
./DataCurator.sif -r myrecipe.toml
```

A more complex example

In full_api.toml you can see an example of how you can specify an entire image processing pipeline with a simple recipe.

```
...
actions=[
        {name_transform=["tolowercase"],
        content_transform=[
                        ["slice_image", [1,2],[[20,50],[20,50]]],
                        ["gaussian", 3],
                        "laplacian",
                        ["threshold_image", "abs <", 0.01],
                        ["apply_to_image", ["abs"]],
                        ["apply_to_image", ["log"]]
                        "otsu_threshold_image",
                        "erode_image"
                        ],
                        mode="copy"}
        ]
...
```

Part II

Installation

1.2 Installation

Installation depends on what you plan to do, and what is available to you.

You can install DataCurator in 4 ways:

- as a Julia package in your global Julia environment
  - assumes you have Julia
- as a local package / cloned repository (no change to global)
  - assumes you have Julia and git
- download a container / executable image **recommended**
  - assumes you have a command line interface (WSL on Windows, any Linux or Mac)
- As a Singularity image
  - assumes you have Singularity installed
  - assumes you have a command line interface (WSL on Windows, any Linux or Mac)

If you do not intend to write code, pick the container/image option

The container comes with Julia, so you don't need to install anything, and it has an optimized precompiled image of DataCurator inside so the startup time reduces to < 1s.

Code snippets

When we include a snippet like:

```
julia
julia>
```

The 2nd `julia>` indicates how the user prompt has changed. It should not be copy-pasted. We do this only where it can help you to see what you should expect to see.

As a Julia package

You need:

- Julia

```julia
using Pkg;
Pkg.add(url="https://github.com/bencardoen/DataCurator.jl")
Pkg.test("DataCurator")
using DataCurator
```

or interactively

```
julia> ] # typing right bracket opens package manager
pkg 1.x> add https://github.com/bencardoen/DataCurator.jl
pkg 1.x> test DataCurator
```

Julia's package manager

Julia's package manager can be invoked with Julia commands, but also responds to interactive use in the REPL (Julia command line).

```
]
<cmd> <argument>
# is the same as
using Pkg; Pkg.cmd(argument)
```

Where cmd is usually one of `test`, add, `activate`, ...

Note: when this repo is private this will prompt for username and github token (not psswd)

As a local repository

You need:

- Julia
- git

```
git clone git@github.com:bencardoen/DataCurator.jl.git ## Assumes ssh
# git clone https://github.com/bencardoen/DataCurator.jl.git ## For non SSH
cd DataCurator.jl
julia
julia>using Pkg; Pkg.activate("."); Pkg.instantiate(); Pkg.test();
```

Github's switch to tokens

Recently Github has been noted to favor token based logins, that won't matter if this repository is public, but while it's not, and you do not use SSH keys, ensure you switch from password to token.

As an executable image

You need:

- A command line environment (WSL on windows, any shell on Linux or MAC)

You'll download an `image`, which is a stand alone environment we prepare for you, with all dependencies. You don't need Julia, nor Singularity, but you do need some way of interacting with it on the command line.

See Sylabs for up to date images.

As a Singularity container

You need:

- A command line environment (WSL on windows, any shell on Linux or MAC)
- Singularity

```
singularity pull --arch amd64 library://bcvcsert/datacurator/datacurator_f35_j1.6:0.0.1
```

The container provides:

- Fedora 35 base environment
- Julia 1.6.2 base installation
- DataCurator installed in its own environment at /opt/DataCurator.jl

Test

If you want to verify everything works as it should:

```
using Pkg;
Pkg.test("DataCurator")
```

or if you cloned the repository

```
using Pkg;
Pkg.activate('.')
Pkg.test('.')
```

Advanced users

Changing the image See buildimage.sh and singularity1p6.def on how the images are built if you want to modify them.

Speeding up start-up time On first run Julia needs to compile functions and load packages. If you process large datasets, this cost (up to 20s) is meaningless. However, for smaller use case its starts to get annoying.

We avoid this cost by using [PackageCompiler.jl] by

- run a typical example of DataCurator so Julia sees which functions are common
- precompile all major dependencies into a system image
- tell Julia to use that image instead.

This is automated in the Singularity image, but for completeness:

```
julia --project=. src/mktest.jl
julia --project=. --trace-compile=dc_precompile.jl src/curator.jl -r example_recipes/
    aggregate_new_api.toml
julia --project=. src/setupimage.jl
```

Now when you want to run DataCurator, do:

```
julia --project=/opt/DataCurator.jl --sysimage /opt/DataCurator.jl/sys_img.so /opt/DataCurator.jl/
    src/curator.jl --recipe <YOURRECIPE.TOML>
```

Linking with Slack

In a Slack workspace, go to Preferences / Manage Apps / Build Create a new App, then configure a webhook (and a channel). That should give you an URL of the form

```
/services/<code>/<code>/<code>
```

Save this is a file

```
echo "/services/<code>/<code>/<code>" > endpoint.txt
```

Then at execution, pass it to DataCurator

```
julia --project=. scripts/monitor.jl -r recipe.toml -e endpoint.txt
```

That's all. On finishing execution, it will print a summarized message to your slack channel of choice.

Part III

Quikstart

Chapter 2

Usage

This chapter details how to quickly get started using DataCurator, either in recipe (text only) mode, or using the Julia API.

2.1 Using recipes only

```
./DataCurator.sif -r myrecipe.toml [---verbose]
```

or a bit more advanced:

```
singularity exec DataCurator.sif julia --project=/opt/DataCurator.jl --sysimage /opt/DataCurator.jl/
    sys_img.so /opt/DataCurator.jl/src/curator.jl --recipe myrecipe.toml
```

You can see why we made the executable image with the very short command, right?

However, it can be useful to explore the package more inside the singularity image

```
singularity exec DataCurator.sif julia <your script>
```

You can also open a shell inside the image

```
singularity shell DataCurator.sif
singularity>julia
julia 1.x>
```

2.2 Recipes + Julia

Either run this in the image, or with the package

```julia
using DataCurator
result = create_template_from_toml("recipe.toml")
if ~isnothing(result) # result will be nothing if something went wrong creating your template
  c, t = res
  counters, lists, returnvalue = delegate(c, t)
end
```

You can next iterate over the counters or lists, if needed. Note that aggregation operations at that point have completed.

```
for counter in counters
    @info counter
end
```

See the API reference for full details.

2.3 Using the Julia API

When you can write Julia you can do anything the template recipes allow and extend it with your own functions, compile more complex functions, and so forth. In this section we'll walk you through how to do this.

Typesafe templates

We heavily use Julia's multiple type dispatch system, so when you make a template

```
template = [mt(is_tif_file, show_warning)]
```

it is internally transformed to a named tuple

```
template[1].condition == is_tif_file # true
template[1].action == show_warning
```

As a user this may not be relevant to you, but it does help in simplifying the code and optimizing the execution quite dramatically. The Julia compiler for example knows the difference at compile time between

```
fs = [is_tif_file, show_warning, quit]
A = mt(fs...) # condition, action, counteraction
fs = [is_tif_file, show_warning]
B = mt(fs...) # condition, action
```

A and B are resolved at compile time, not at runtime, improving execution speed while ensuring type safety.

To put it differently, your template is usually precompiled, not interpreted, as it would be in bash/Python scripts.

Examples

Replace whitespace and uppercase

Rename all files/directories with ' ' in them to '_' and switch any uppercase to lowercase.

```
condition = x -> is_upper(x) | has_whitespace(x)
fix = x -> whitespace_to(lowercase(x), '_')
action = x -> transform_inplace(x, fix)
transform_template(rootdirectory, [mt(condition, action)]; act_on_success=true)
```

Next, we verify our dataset has no uppercase/whitespace in names.

```
count, counter = generate_counter()
verify_template(rootdirectory, [mt(condition, counter)]; act_on_success=true)
@info count
```

Flatten a file hierarchy

```
action = x->flatten_to(root, x, newdirectory)
verify_template(root, [mt(isfile, action)]; act_on_success=true)
```

Extract all 3D tif files to a single directory

Note, this will halt if the images it extracts exist on the target directory.

```
trigger = x-> is_3d_img(x)
action = x->flatten_to(root, x, image_directory)
verify_template(root, [mt(trigger, action)]; act_on_success=true)
```

Sort 2D image and csv files into separate directories

```
img_action = x->flatten_to(root, x, image_directory)
csv_action = x->flatten_to(root, x, csv_directory)
verify_template(root, [(mtis_2d_img, img_action), (is_csv_file, csv_action)]; act_on_success=true)
```

Compute size in bytes of a large hierarchy in parallel

```
count, counter = generate_size_counter()
verify_template("rootdirectory", [mt(isfile, counter)]; act_on_success=true)
@info "Size of matched files = $(count) bytes"
```

Compute size of 2 vs 3D images separately

```
count2, counter2 = generate_size_counter()
count3, counter3 = generate_size_counter()
verify_template("rootdirectory", [mt(is_2d_img, counter2),(is_3d_img, counter3)];
↪  act_on_success=true)
@info "$(count2) bytes of 2D images, $(count3) of 3D images"
```

Hierarchical recipes

Suppose your data is supposed to have this layout

```
- root
  - replicate nr
    - celltype
      - cell nr : of the form "Series XYZ"
        - 2 tif files, 3D, ending with 1,2.tif
```

You can use **hierarchical** templates, that give you very precise control of where a condition fires

Create a hierarchical template

```
onfail = x->show_warning
template = Dict()
```

First, define what to do with unexpected directories/files

```
template[-1] = [mt(never, onfail)]
```

never is a shortcode symbol for 'will never pass'

At root, we only expect sub directories

```
template[1] = [mt(isdir, onfail)]
```

Replicate should be integer

```
template[2] = [mt(x->all_of(x, [isdir, integer_name]), onfail)]
```

Celltype should only be subdirs

```
template[3] = [mt(isdir, onfail)]
```

Lowest data directory should end with cell nr, and have 2 or more files

```
inputdir_check = x->all_of(x, [isdir, x->ends_with_integer(x), x->n_files_or_more(x, 2)])
template[4] = [mt(inputdir_check, onfail)]
```

Actual files should be 3D images

```
file_check = x -> is_tif_file(x) & is_3d_img(x) & endswith(x, r"[1,2].tif")
template[5] = [mt(file_check, onfail)]
```

Execute

```
verify_template(root, template; traversalpolicy=topdown, parallel_policy="parallel")
```

Advanced You are free to define even more complex actions, for example, triggers that fire on invalid data AND valid data in 1 template. For example, let's say we expect csv files, and if we find tif files, then we delete those, otherwise we just warn.

```
template[y] = [mt(x->~is_tif_file(x), delete_file), mt(is_csv_file, onfail)]
```

2.4 Fire triggers when they are true, not when they fail

If it's hard to define conditions that should succeed, you can reverse the firing conditions, but more easily readable is just asking the verifier to do so for you

```
verify_template(root, template; act_on_succes=true)
```

Parallel execution

All recipes can be executed in parallel. Counters are protected so they are threadsafe, yet need no locks.

```
count, counter = generate_counter(true; incrementer=size_of_file)
verify_template("rootdirectory", [mt(condition, counter)]; parallel_policy="parallel",
↪  act_on_success=true)
@info "Size of matched files = $(count) bytes"
```

Part IV

Documented recipe with all features demonstrated

Chapter 3

A recipe using all/most of the possible features

3.1 Recipes

First, a recipe is a plain text file, in TOML format, designed to be as human friendly as possible.

We'll run through all, or most of the features you can use, with example TOML snippets.

Any `recipe` needs 2 parts

- the global configuration
- the actual template.

The global configuration specifies **how** the template is applied, the template specifies the conditions/rules to apply, i.o.w. the **what** and **when**.

A *section* in a TOML file is simply:

```
[mysectionname]
mycontent="some value"
```

3.2 Global section

The smallest possible global section looks like this:

```
[global]
inputdirectory="where/your/data/is/stored"
```

Next, we can either act on failure (usually in validation), or on success. This simply means that, if set to false, we check for any data that **fails** the rule you specify, then execute your actions. In datacuration you'll want the inverse, namely, act on success.

> **You can have your cake and eat it**
>
> You can specify `actions` AND `counter_actions`, allowing you to specify what to do if a rule applies, and what if it doesn't. In other words, you have maximal freedom of expression.

```
act_on_success=false # default
```

We can also specify how we traverse data, from the deepest to the top (`bottomup`), or `topdown`. If you intend to modify files/directories in place, `bottomup` is the safer option.

```
traversal="bottomup" # or topdown
```

We can validate or curate data in parallel, to maximize throughput. Especially on computing clusters this can speed up completion time. If true, will use as many threads as the environment variable **JULIA*NUM*THREADS** (usually nr of HT cores).

```
export JULIA_NUM_THREADS=2
```

Thread safety

You do not need to worry about `data races`, where you get non-deterministic or corrupt results, if you stick to our conditions and aggregations, there are no conflicts between threads.

```
parallel=true #default false
```

By default your rules are applied without knowing how deep your are in your dataset. However, at times you will need to know this, for example, to verify that certain files only appear in certain locations, or check naming patterns of directories. For example, a path like `top/celltype/cellnr` will have a rule to check for a cell number (integer) at level 3, not anywhere else. To enable this:

```
# If true, your template is more precise, because you know what to look for at certain levels [
     level_i]
# If false, define your template in [any]
hierarchical=true
```

For more complex pattern matching you may want to use Regular Expressions (regex), to enable this:

```
# If true, functions accepting patterns (endswith, startswith), will have their argument converted
     to a regular expression (using PRCE syntax)
regex = false
```

The inputdirectory should point to your dataset. The outputdirectory is where global output is written, e.g. output of aggregation.

```
inputdirectory=...
outputdirectory=...
```

Slack support

If you want to configure/use Slack based actions, enable slack support by pointing the global configuration to an endpoint file. This should contain 1 line of the form

```
 /services/<code>/<code>/<code>
```

See installation for how to set this up.

```
endpoint=endpointfile.txt
```

At the end of the template, DC will print a summary to the slack channel of your choice with status, time, and counters.

You can also use slack based actions, see example_recipes/slack.toml

Saved actions and conditions

Quite often you will define actions and conditions several time. Instead of repeating yourself, you can define actions and conditions globally, and then refer from your template to them later. For example:

```
common_actions = {react=[["all", "show_warning", ["log_to_file", "errors.txt"], "remove"]]}
common_conditions = {is_3d_channel=[["all", "is_tif_file", "is_3d_img", "filename_ends_with_integer
    "]]}
```

In your template you can then do

```
actions=["react"]
```

instead of

```
actions=[["all", "show_warning", ["log_to_file", "errors.txt"], "remove"]]]
```

This is useful because: - default actions/conditions are more concisely expressed and reused - composing complex rules without running out of screen real estate - more legible - if you want to change a complex rule, you only need to do so in 1 place - for Julia, instead of multiple executable rules, there's now 1

The reference syntax is

```
common_..={name1=[["all", f1, f2, f3, ...]], name2=...}
```

Where f1, f2, ... are conditions/actions, and `name1` will be a placeholder you can reference later to.

> **Nested [[]]**
>
> Here you need to use the explicit nested form for anything more than 1 action/condition, because `all=true` is implied. Note that this section is parsed before the template itself is seen at all.

!!! warning Common actions/conditions cannot refer to others when you're defining them. If this was possible, we'd run the risk of deadlock, where actions refer to themselves in a loop, for example. If you need this kind of functionality, it's better to use the Julia API.

Aggregation

Aggregation is a complex word for use cases like:

- counting files matching a pattern
- counting total size of a selection of files
- making lists of input/output pairs for pipelines
- combining 2D images into 1 3D image
- combining 2D images, sorted by prefix (e.g. 'abc*1.tif', 'abc*2.tif', 'cde*1.tif', 'cde*2.tif' -> abc.tif, cde.tif)
- selecting specific columns from each csv you find, and fusing all in 1 table
- finding files that match a pattern, sort them, find only unique ones, and then save them in a file or table

You can do any of these all at the same time with `counters` and `file_lists` in the global section:

```
counters = ["filecounter", ["sizecounter", "size_of_file"]]
```

Here we created 2 simple counters, one that is incremented whenever you refer to it, and one that when you pass it a file, records it total size in bytes. When the program finishes, these counters are printed, but also saved as counters.csv.

To refer to these, you can do the following

```
actions=[["count", "filecounter"], ["count", "sizecounter"]]
```

At the end you would have a dataframe/csv such as:

```
name          | count
filecounter   | 1024
sizecounter   | 1230495
```

File aggregation The simplest kind just adds a file each time you refer to it, and writes them out in traversal order (per thread if parallel) at the end to "infiles.txt"

```
file_lists = ["infiles"]
```

To make input-output pairs you'd do

```
file_lists = ["infiles", ["outfiles", "outputpath"]]
```

Let's say we add a file "a/b/c.txt" to infiles, when we add it to outfiles it will be recorded as: "/outputpath/a/b/c.txt" This is a common use case in preparing large batch scripts on SLURM clusters.

What if we want to collect files or paths, but instead of collecting them in order of traversal (discovery), we want to sort them first, and only keep the path, not the filenames.

```
file_lists = [{name="mylist", aggregator=[["filepath",
                                          "sort",
                                          "unique",
                                          "list_to_file"]]},
```

So the following

```
/a/b/1/1.csv
/a/b/1/2.csv
/a/b/2/1.csv
/a/b/2/2.csv
/a/b/2.csv
```

would be written as a `mylist.txt` containing

```
/a/b/1
/a/b/2
/a/b
```

Image aggregation

```
file_lists = [{name="3dstack.tif", aggregator="stack_images"}]
file_lists = [{name="3dstack.tif", transformer=["reduce_images", ["maximum", 2]],aggregator="
     stack_images"}]
```

```
file_lists = [{name="image_stats", transformer=["describe_image", 3], aggregator="concat_to_table"}]
```

For each image added to the list, it'll slice the image along the z axis and create a table with statistics on intensity (min, mean, std, kurtosis, Q1, ...), for example:

```
|
  Row │ minimum     Q1        mean      median    Q3        maximum   std       kurtosis  slice
     axis   source|
       │ Float64     Float64   Float64   Float64   Float64   Float64   Float64   Float64   Int64
     Int64  String7
     |─────┼──────────────────────────────────────────────────────────────────────────────────────────────────|

    1 │ 0.00784314  0.245098  0.508418  0.501961  0.760784  0.996078  0.291711   6.71003      1
      3  1.tif|
    2 │ 0.00392157  0.242157  0.490539  0.482353  0.741176  1.0       0.290982   6.60052      2
      3  1.tif
...
```

Stack images, sorting by prefix Sometimes image datasets have files like

```
root├──
 patient1|
   ├── patient1_slice_1.tif|
   └── patient1_slice_2.tif|
   └── ...├──
 patient2|
   ├── patient2_slice_1.tif|
   └── patient2_slice_2.tif|
   └── ...
...
```

We'd like to combine these into

```
- patient1.tif (3D)
- patient2.tif (3D)
```

The solution is straightforward, we aggregate but ask to group by prefix

```
file_lists = [{name="slices", aggregator="stack_images_by_prefix"}]
```

```
file_lists = [{name="all_ab_columns.csv", transformer=["extract_columns", ["A", "B"]], aggregator="
     concat_to_table"}]
```

or if you want to aggregate columns first

```
file_lists = [{name="all_ab_columns.csv", transformer=["groupbycolumn", ["x1", "x2"], ["x3"], ["sum
     "], ["x3_sum"]], aggregator="concat_to_table"}]
```

Template

A template has 2 kind of entries `[any]` and `[level_X]`. You will only see the level_X entries in hierarchical templates, then X specifies at which depth you want to check a rule.

Flat templates, the Any section

```
[any]
all=false #default, if true, fuses all conditions and actions. If false, you list condition-action
     pairs.
conditions=["is_tif_file", ["has_n_files", 10]]
actions=["show_warning", ["log_to_file", "decathlon.txt"]]
counter_actions=[["add_to_file_list", "mylist"], ["log_to_file", "not_decathlon.txt"]] ## Optional
```

The `add_to_file_list` will pass any file or directory for which `is_tif_file` = true (see `act_on_success`) to a list you defined earlier called "mylist". You specified in the global section what needs to be done with those files at the end. You do not need counter_actions.

Negation and logical and

You can also negate and fuse conditions/actions. Actions can not be negated.

```
conditions=[["not", "is_tif_file"], ["all", "is_2d_img", "is_rgb"]]]
```

This is useful if you want to check for multiple things, but each can be quite complex. In other words, you want pairs of condition-action, so all=false, yet each pair is a complex rule.

Aliases

`add_to_file_list` is aliased to `aggregate_to`, use whichever makes more sense in reading the recipe.

Hierarchical templates, with `level_X`

All you now need to add is what to do at level 'X'

```
[global]
hierarchical=true
...
[level_3]
conditions=...
actions=...
...
```

This will only be applied if, and only if, files and directories 3 levels (directories) deep are encountered.

Sometimes you do not know how deep your dataset can be, in that case you'll want a 'catch-all', in hierarchical templates this is now the role of any

```
[global]
act_on_success=true
[any]
conditions=["is_csv_file"]
actions=["show_warning"]
[level_3]
conditions=["is_tif_file", "is_csv_file"]
actions=[["log_to_file", "tiffiles.txt"], "show_warning"]
```

This tiny template will write any tif file to tiffiles.txt. If it encounters csv files anywhere else, it will warn you.

Please see the directory example_recipes for more complex examples.

Advanced usage

Verify a complex, deep dataset layout. This is an example of a real world dataset, with comments.

An example of a 'curated' dataset would look like this

```
root├──
 1                           # replicate number, > 0│
   ├── condition_1             # celltype or condition  # <- different types of GANs for generating
     data│
   │   ├── Series005           # cell number│
   │   │   ├── channel_1.tif   # first channel, 3D Gray scale, 16 bit│
   │   │   ├── channel_2.tif   # second channel, 3D Gray scale, 16 bit
...├──
 2
...
```

Let's create a recipe for this dataset that simply warns for anything unexpected.

Global section We're validating data, so we'll specify what should be true, and only if our rules are violated, do we act. Hence `act_on_success=false`, which is the default. We have different rules depending on where in the hierarchy we check, so `hierarchical=true`. And finally, we need a place to start, so `inputdirectory=root`

```
[global]
hierarchical=true
inputdirectory = "root"
```

The template We specify what to do if we see anything that does not catch our (5-level) deep recipe, in the `[any]` section.

```
[any]
conditions=["always_fails"]        #if this rule ever is checked, say at level 10, it fails
     immediately
actions = ["show_warning"]
```

Next, we define rules for each level. Levels:

```
## Top directory 'root', should only contain sub directories

[level_1]
```

```
conditions=["isdir"]
actions = ["show_warning"]   # if we see a file, isdir->false, so show a warning
## Replicate directory, should be an integer

[level_2]
all=true
conditions=["isdir", "integer_name"]  # again, no files, only subdirectories
actions = ["show_warning"]

## We don't care what cell types are named, as long as there's not unexpected data

[level_3]
conditions=["isdir"]
actions = ["show_warning"]

## Final level, directory with 2 files, and should end with cell nr
[level_4]
all=true
conditions=["isdir", ["has_n_files", 2], ["ends_with_integer"]]
actions = ["show_warning"]

## The actual files, we complain if there's any subdirectories, or if the files are not 3D
[level_5]
all=true
conditions=["is_tif_file", ["endswith", "[1,2].tif"], ["not", "is_rgb"], "is_3d_img",]
actions = ["show_warning"]
```

Short circuit to help to speed up conditions

Note that we first check the file extension `is_tif_file`, and only then check the pattern `endswidth ...`, and only then actually look at the image type. Checking if an image is 3D or RGB requires loading it. Loading (potentially huge) files is slow and expensive, so this could mean we'd check 'is*3d*img' for a csv file, which would fail, but in a very expensive way. Instead, our conditions `short circuit`. We specified `all=true`, so each of them has to be true, if 1 fails we don't need to check the others. By putting `is_tif_file` first, we avoid having to even load the file to check its contents. This is done **automatically** for you, as long as you keep to the left-right ordering, in general of cheap(or least strict) to expensive (most strict). In practice for this dataset, this means a runtime gain of 50-90% depending on how much invalid data there is.

Early exit: sometimes you want the validation or processing to stop immediately based on a condition, e.g. finding corrupt data, or because you're just looking for 1 specific type of conditions. This can be achieved fairly easily, illustrated with a trivial example that stops after finding something other than .txt files.

```
[global]
act_on_success = false
inputdirectory = "testdir"
[any]
all = true
conditions = ["isfile", ["endswith", ".txt"]]
actions = ["halt"]
```

Regular expressions: For more advanced users, when you write "startswith" "*.txt", it will not match anything, because by default regular expressions are disabled. Enabling them is easy though

```
[global]
regex=true
...
condition = ["startswith", "[0-9]+"]
```

This will now match files with 1 or more integers at the beginning of the file name.

!!! note Regex compilation errors on "*patterns*" *If you try to pass a regex such as* ".txt", you'll get an error complaining about PCRE not being able to compile your Regex. The reason for this is the lookahead/lookback functionality in the Regex engine not allowing such wildcards at the beginning of a regex. When you write " *.txt* "*, what you probably meant was* '*anything with extension txt*'*, but not the name* "*.txt*"*, which* " **.txt* " *will also match. Instead, use* ".\.txt". When in doubt, don't use a regex if you can avoid it. Similar to Kruger-Dunning, those who believe they can wield a regex with confidence, probably shouldn't.

Negating conditions: By default your conditions are 'OR'ed, and by setting all=yes, you have 'AND'. By flipping action*on*succes you can negate all conditions. So in essence you don't need more than that for all combinations, but if you need to specifically flip 1 condition, this will get messy. Instead, you can negate any condition by giving it a prefix argumet of "not".

```
[global]
act_on_success = true
inputdirectory = "testdir"
regex=true
[any]
all=true
conditions = ["isfile", ["not", "endswith", ".*.txt"]]
actions = [["flatten_to", "outdir"], "show_warning"]
```

Counteractions: When you're validating you'll want to warn/log invalid files/folders. But at the same time, you may want to do the actual preprocessing as well. This is where counteractions come in, they allow you to specify

- Do x when condition = true
- Do y when condition = false

A simple example, filtering by file type:

```
[global]
act_on_success=true
inputdirectory = "testdir"
[any]
conditions=["is_csv_file"]
actions=[["log_to_file", "csvs.txt"]]
counter_actions = [["log_to_file", "non_csvs.txt"]]
```

or another use case is deleting a file that's incorrect, while transforming correct files in preparation for a pipeline, in 1 step.

Export to HDF5/MAT You can save/export directly to HDF5 and MAT, so if you're curating a dataset consisting of files, but your pipeline (for good reason) works on HDF5, you can do so easily.

```
[global]
...
[any]
conditions = ["is_tif_file", "is_csv_file"]
actions=[["add_to_hdf5", "img.hdf5"], ["add_to_mat", "csv.mat"]]
```

Note

The filename will be used as entry/variable in the MAT or HDF5 file, e.g. file->content.

3.3 Modifying files and content

When you want precise control over what function runs on the content, versus the name of files, you can do so. This example finds all 3D tif files, does a median projection along Z, then masks (binarizes) the image as a copy with original filename in lowercase.

```
[global]
act_on_success=true
inputdirectory = "testdir"
[any]
conditions=["is_3d_img"]
actions=[{name_transform=["tolowercase"], content_transform=[["reduce_image", ["maximum", 2]], "mask
     "], mode="copy"}]
```

The examples so far use `syntactic sugar`, they're shorter ways of writing the below, but in certain case where you need to get a lot done, this full syntax is more descriptive, and less error prone. It also gives DataCurator the opportunity to save otherwise excessive intermediate copies.

The full syntax for actions of this kind:

```
actions=[{name_transform=[entry+], content_transform=[entry+], mode="copy" | "move" | "inplace"}+]
```

Where `entry` is any set of functions with arguments. The + sign indicates "one or more". The | symbol indicates 'OR', e.g. either copy, move, or inplace.

Select rows from CSVs and save them

```
[global]
act_on_success=true
inputdirectory = "testdir"
[any]
all=true
conditions=["is_csv_file", "has_upper"]
actions=[{name_transform=["tolowercase"], content_transform=[["extract", ("Count", "less", 10)]],
     mode="copy"}]
```

Table extraction has the following syntax:

```
["extract", (col, op, vals)]
```

or

```
["extract", (col, op)]
```

For example:

```
["extract", ("name","=","Bert"),  ("count", "<", 10)]
```

Gives you a copy of the table with only rows where name='Bert' and count<10.

List of operators:

```
less, leq, smaller than, more, greater than, equals, euqal, is, geq, isnan, isnothing, ismissing,
↪  iszero, <, >, <=, >=, ==, =, in, between, [not, operator]
```

The operators in and between expect an array of values:

```
('count', 'in', [2,3,5])
```

and

```
('count', 'between', [0,100])
```

where the last is equivalent, but shorter (and faster) than:

```
('count', '>', 0), ('count', '<', 100)
```

Aggregation

When you need group data before processing it, such as collecting files to count their size, write input-output pairs, or stack images, and so forth, you're performing a pattern of the form

```
output = reduce(aggregator, map(transform, filter(test, data)))
```

Sounds complex, but it's intuitive, you

- collect data based on some condition (filter)
- transform it in some way (e.g. mask images, copy, ...)
- group the output and reduce it (all filenames to 1 file, ...)

Examples of this use case:

- Collect all CSV files, concat to 1 table
- Collect columns "x2" and "x3" of CSV files whose name contains "infected_C19", and concat to 1 table
- Collect all 2D images, and save to 1 3D stack
- Collect all 3D images, and save maximum/minimum/mean/median projection

The 2nd example is simply:

```
[global]
...
file_lists=[{name="group", transformer=["extract_columns", ["x2", "x3"]], aggregator="
     concat_to_table"}]
...
[any]
all=true
conditions=["is_csv_file", ["contains", "infected_C19"]]
actions=[["add_to_file_list", "group"]]
```

The maximum projection of 2D images

```
[global]
...
file_lists=[{name="group", aggregator=["reduce_images", "maximum"]}]
...
[any]
conditions=["is_2d_img"]
actions=[["add_to_file_list", "group"]]
```

The complete grammar:

```
file_lists=[{name=name, transformer=identity, aggregator=shared_list_to_file}+]
```

(X+) indicates at least one of X

The following aliases save you typing:

```
file_lists=["name"]
# is the same as
file_lists=[{name=name, transformer=identity, aggregator=shared_list_to_file}]
```

```
file_lists=[["name", "some_directory"]]
# is the same as
file_lists=[{name=name, transformer=change_path, aggregator=shared_list_to_file}]
```

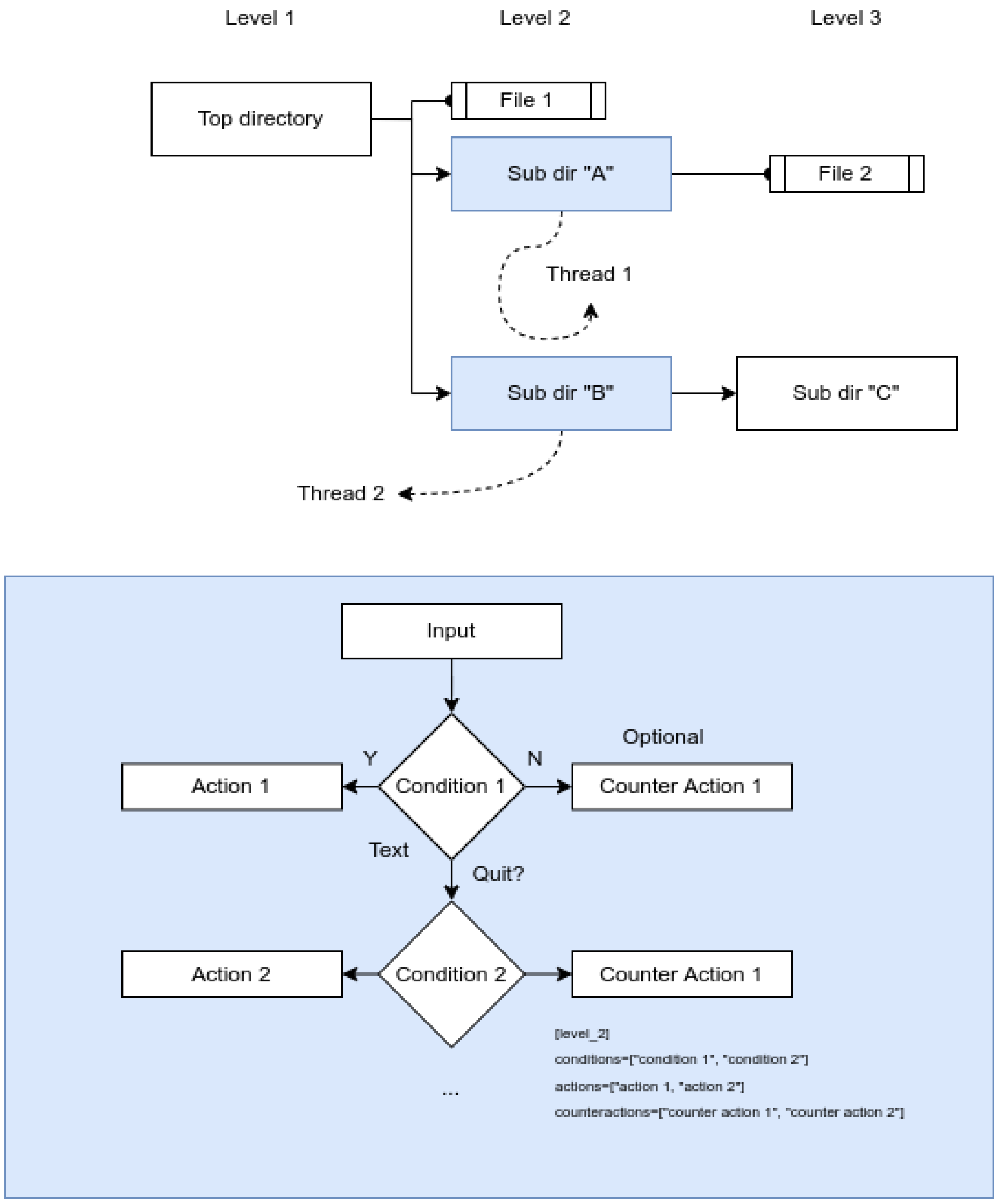
You're free to specify as many aggregators as you like.

3.4 Under the hood

When you define a template, a 'visitor' will walk over each 'node' in the filesystem graph, testing any conditions when appropriate, and executing actions or counteractions.

In the background there's a lot more going on

- Managing threadsafe data structures
- Resolving counters and file lists
- Looking up functions
- Composing functions and conditions
- ...



```
[level_2]
conditions=["condition 1", "condition 2"]
actions=["action 1, "action 2"]
counteractions=["counter action 1", "counter action 2"]
```

Figure 3.1: Concept

Part V

Conditions and Actions for use in recipes

Chapter 4

Actions and Conditions you can use in recipes

Don't repeat yourself

Quite often you will want to apply certain conditions or actions several times in a hierarchical template. In that case you can define `common_actions` and `common_condtions` in the `[global]` section, which you can refer to by name. Any of the actions and conditions below can be used to compose more complex actions and conditions.

4.1 Actions

File/folder name:

```
whitespace_to  usage: ["whitespace_to", "_"]
quit
proceed
filename
show_warning
quit_on_fail
log_to_file
ignore
sample
size_of_file
remove
delete_file
delete_folder
path_only
filepath
show_warning
log_to_file_with_message
remove_from_to                    usage: ["remove_from_to", "from_pattern", "to_pattern"], see
↪  example_recipes/remove_pattern.toml
remove_from_to_extension_inclusive
remove_from_to_extension_exclusive
remove_from_to_exclusive
remove_from_to_inclusive
remove_pattern
replace_pattern
read_postfix_int
read_prefix_int
read_int
read_postfix_float
```

```
read_prefix_float
read_float
```

Aggregation

When you use aggregation to combine files into lists, it can be helpful to transform filenames in a group, for example, ensuring only unique files are written to file, or they're sorted, rather than file traversal order.

> **Aggregation 101**
>
> You specify in the `[global]` section, in the entry `file_list=...` what the name of a list is, and what, if any, needs to be done with the list of files, be it concatenate csvs to a table, reduce images, describe images, sort, ... Once defined, you refer to them by name in actions.

You can use the following actions

```
actions=[["add_to_file_list", "listname"]]
# or
actions=[["add_to_file_list", ["listname", "listname2"]]
```

The second allows you to add a single file to multiple aggregation lists. The following all do the same, but are defined to be used if they're more readable this way.

```
actions=[["aggregate_to", "listname"]]
actions=[["->", "listname"]]
```

Example

```
[global]
act_on_success=true
file_lists = [{name="table", aggregator=[["filepath",
                                          "sort",
                                          "unique",
                                          "shared_list_to_file"]]},
              {name="out", aggregator=[[["change_path", "/tmp/output"],
                                         "filepath",
                                         "sort",
                                         "unique",
                                         "shared_list_to_file"]]}
              ]
inputdirectory = "testdir"
[any]
all=true
conditions = ["is_csv_file"]
actions=[["add_to_file_list", "table"], ["add_to_file_list", "out"]]
```

This example collects all csv files, records only the path, not the file name, and creates 2 lists, in input/output pairs. For example for files:

```
/a/b/c/1.csv
/a/b/c/2.csv
```

You will get two files:

```
#table.txt
/a/b/c  # The sorted unique path to 1, 2.csv
```

and

```
#out.txt
/tmp/output/a/b/c  # The sorted unique path to 1, 2.csv linked to new output directory
```

This can be useful when you're generating input / output lists for batch processing, where you pipeline expects to see a directory with csv files, and wants to write output to an equivalent location starting at a different path. (e.g. SLURM array jobs)

Content:

Image operations

These operations fall into 3 categories:

- Increase dimension, e.g. 10 2D images to 1 3D
- Decrease/reduce dimension, e.g. 10 2D images to 1 2D, or 1 3D to 1 2D
- Keep dimension, but change voxels, but not dimension, e.g. mask, filter, ...
- Keep dimension, but reduce size: image slicing

```
stack_images
```

A special case:

```
stack_images_by_prefix
```

This assumes you have files with a pattern like `A_1.tif, A_2.tif, ..., B_1.tif`. If each has K dimensions, you'll end up with 1 file per prefix (here 2), with K+1 dimensions.

See the aggregation section for details.

N to N-1 dimension: For aggregation (combine many images in N x 2D to 1 x 2D):

```
reduce_images
#usage
["reduce_images", ["maximum", 3]] for max projection on Z
```

For per image reduction (1 image, 3D -> 2D):

```
reduce_image + maximum, minimum, median, mean + dim : 1-N
# example:
["reduce_image" ,["maximum", 2]]
```

N to N The image dimensions stays the same, but the voxels are modified

```
mask
laplacian
gaussian, sigma
image_opening
image_closing
erode_image
dilate_image
```

```
threshold_image, operator, value
otsu_threshold_image
```

where operator can be any of "<", ">", "=", "abs operator"

For example

```
"threshold_image", "abs >", 0.2
```

Sets all voxels where the magnitude (unsigned) > 0.2 to 0.

`otsu_threshold` computes the threshold automatically.

```
slice_image, dimension, slice
slice_image, dimension, slice_from, slice_to
slice_image, [dimensions], [slices]
```

For example

```
"slice_image", [1,3], [[200,210],[1,200]]
```

which is equivalent to

```
img[200:210,:,1:200]
```

!!! warning Indexing Julia indices into array, image, tables, start at **1**, not 0. This is similar to Matlab, but unlike C/Python. Dimension=1 refers to the X-axis, and so forth.

Image statistics

To get a CSV table of statistics of the image intensity distribution you can do

```
describe_image[, axis]
```

Without axis 1 row per image is produced, with axis the distribution is computed along gives axis. Example:

```
   minimum    Q1        mean      median    Q3        maximum  std       kurtosis  axis   source
     slice
   Float64    Float64   Float64   Float64   Float64   Float64  Float64   Float64   Int64  String
     Int64│
  0.0156863  0.258824  0.485621  0.454902  0.730392      1.0  0.283031    6.6581      0  1.tif
      1
```

To describe objects in an image (assuming it's thresholded or can be binarized) (3D only)

```
describe_objects
```

Example

```
size     weighted  minimum    Q1        mean      median    Q3        maximum  std       kurtosis
     xyspan   zrange   zmidpoint  filename
Float64  Float64   Float64    Float64   Float64   Float64   Float64   Float64  Float64   Float64
     Float64  Float64  Float64    String
216.0   104.894  0.0156863  0.258824  0.485621  0.454902  0.730392      1.0  0.283031    6.6581
     8.48528      6.0       23.0    ...
```

You can use this in aggregation, for example, to describe all objects in all channel 1 tifs

```
[global]
...
file_lists = [{name="objects", aggregator=[["describe_objects",
                                          "concat_to_table"]]},]
...
[any]
all=true
conditions = ["is_tif_file"]
actions=[[""aggregato_to", "objects"]]
```

Table operations

```
concat_table
```

```
extract_columns  usage: ["extract_columns", ["x1", "x2"]]
```

```
[command, [[columnname, operator, arguments],...]]
# or
[command, [columnname, operator, arguments]]
```

where 'command' is one of `extract`, `delete`.

The `operator` is one of:

```
less, leq, smaller than, more, greater than, equals, euqal, is, geq, isnan, isnothing, ismissing,
     iszero, <, >, <=, >=, ==, =, in, between, [not, operator]
```

Warning

Do not use 'col','=','NaN' but `'col', 'isnan'`. Similar for iszero, isnothing, ... Floating point rules specify that Nan!=Nan, so your condition will always be false.

Between To express 1 < a < 2, where a is a column name, you could write

```
["a", ">", 1], ["a","<", 2]
```

You can save yourself typing, and just write:

```
["a", "between", [1, 2]]
```

Warning

Make sure you pass a vector of 2 values!!

in To find all values of a column in a defined set:

```
["a", "in", [2,3,5]]
```

Negating You can also negate an operator, if that makes sense for your use case:

```
["a"], ["not" "in"], [[2,3,5]]
["a"], ["not" "between"], [[1, 2]]
```

4.2 Conditions

Each action can only be applied if a condition fires. This is a list of all conditions you can use, alone, in action-condition pairs, combined (with all=true), or nested:

File/directory name conditions

```
integer_name # file or directory name is an integer, e.g. "2", "003", but not "One" or "_1"
is_lower
is_upper
has_whitespace
has_upper
has_lower
is_hidden[_dir, _file]
has_integer_in_name
has_float_in_name
filename_ends_with_integer
```

Directories

```
isdir/isfile
has_n_files
n_files_or_more
less_than_n_files
has_n_subdirs
less_than_n_subdirs
```

File type checks

These check by file extension, they do NOT open files. Opening a file, or trying to figure out by not failing, is a slow operation compared to checking file extensions. You'll have to decide which is more appropriate, there are `is_img` and variants that do load an image to check.

```
is_csv_file
is_tif_file
is_png_file
has_image_extension
is_type_file # usage : ["is_type_file", ".csv"]
file_extension_one_of # usage : ["file_extension_one_of", [".csv", ".txt", ".xyz"]]
```

Image specific

!!! note Content type testing Testing if a file is an image means passing it to the image library and letting it try loading the file. For large files this can be expensive.

So instead of:

```
is_img
```

it's smarter to do:

```
["is_file", "is_tif_file", "is_img"]
```

!!! note RGB v 3D Julia uses the convention that RGB != 3D, which saves you from a lot of disambiguation. For example, is 10x10x10 32bit an RGB+alpha 3D image? Or just a 32bit Float 3D image? Julia will load the right type from the file, so it's one less worry.

```
is_img    # NOT the same as has_image_extension, this will try to load the file
is_kd_img # usage ["is_kd_img", 3]
is_2d_img
is_3d_img
is_rgb
is_8bit_img
is_16bit_img
```

Checking image dimensions Either to verify correct data layout, or when you're going to slice images, it's handy to be able to check dimensions

```
size_image, [[dimension, operator, limit(s)],..]
```

Example:

```
"size_image", [[1, ">", 4], [2, ">", 4], [3, "between", [1, 1000]]
```

Operators: >, <, =, >=, <=, between, in

Table specific

Table refers here to tabular data contained in CSV files, loading into Julia DataFrames. In short, if it has columns and rows, in a csv, it's a table.

Useful before you concatenate tables:

```
has_n_columns
has_less_than_n_columns
has_more_than_or_n_columns
```

Checking if your table has the right columns:

```
has_columns_named  # usage ["has_columns_named", ["Age", "Heart Rate"]]
```

This checks if those 2 columns are in the table, not if those are the only 2 columns.

General

```
always, never
```

Self-explanatory, sometimes handly:

```
always = x -> true
never = x -> false
```

If you're testing conditions, you can use these as placeholders, for example.

If you're not familiar with Julia, the following are builtin:

```
maximum/minimum/median/mean
size
isfile
isdir
ispath
splitpath
splitdir
splitext
basename
length
sum
isnothing
```

Part VI

API Reference

Chapter 5

Reference

DataCurator.ParallelCounter – Type.

```
Usage
QT = ParallelCount(zeros(Int64, Base.Threads.nthreads()), Int64(0))
QT.data[threadid()] = ...
```

source

DataCurator._printtoslack – Method.

```
Print a message to a connected slack
```

source

DataCurator.apply_to – Method.

```
apply_to(x, f; base=true)

    Where x is a path, if base=false, return f(x), otherwise works on the last part of the path
```

source

DataCurator.apply_to_image – Method.

```
apply_to_image(img, operators::AbstractVector{T}) where {T<:AbstractString}

Apply each of the operators, left to right, to the image (in place.)

Operators can be any unary operators in scope that can be vectorized, e.g. log, sin, cos, abs,
↪  ...
```

source

DataCurator.bottomup – Method.

```
topdown(node, expander, visitor; context=nothing, inner=expand_sequential)
    Recursively apply visitor onto node, until expander(node) -> []
    If context is nothing, the visitor function gets the current node as sole arguments.
    Otherwise, context is expected to contain: "node" => node, "level" => recursion level.
    Inner is the delegate function that will execute the expand phase. Options are :
    ↪  expand_sequential, expand_threaded

    Traversal is done in a post-order way, e.g. visit after expanding. In other words, leaves
    ↪  before nodes, working from bottom to top.
```

source

DataCurator.buildcomp – Method.

~~~
buildcomp(dataframe, column, operator, value)

Dispatches to the correct form of operator.(df[:,col], value)

operator (String) can be one of:
less, leq, smaller than, more, greater than, equals, euqal, is, geq, isnan, isnothing,
↪  ismissing, iszero, <, >, <=, >=, ==

The operator can be negated: ["not", "less"]
Multi-argument comparison are also supported:
- between [x, y]
- in [x, y, z]

You can repeat columns, but non-existing columns are an error.

```julia
df = DataFrame(zeros(2,2),:auto)
df[1,:] .= 5
cols, ops, vals = ["x1", "x1", "x1"], [["not", "in"],"less", ["not", "isnan"]], [[1,2,3,5],10,
↪  "NaN"]
@info reduce(&., buildcomp(df, c, o, v) for (c, o, v) in zip(["x1", "x1", "x1"], )
~~~
```

source

DataCurator.canwrite – Method.

```
canwrite(filename)

tests if `filename` can be opened for writing
```

source

DataCurator.collapse_functions – Method.

```
collapse_functions(fs; left_to_right=false)

Generalization of (f, g) --> x->(f(g(x))) for any set of functions
left_to_right : g(f(x)), otherwise f(g(x))
```

source

DataCurator.create_template_from_toml – Method.

```
create_template_from_toml(tomlfile)

Parse a toml encoded recipe, decode all the actions and conditions, and return a configuration
↪  (Dict) and executable template.

```julia
c, t = create_template_from_toml(tomlfile)
delegate(c, t)
```
```
```

source

DataCurator.decode_dataframe_function – Method.

```
decode_dataframe_function(x::AbstractVector, glob::AbstractDict)

Decodes an entry of the form [command, [(col, op, vals)+]] into a function object for the
↪  template

Both the single tuple version
    command, (col, op, vals)
and longer version
    command, [(col, op, vals), ...]
are valid, and dealt by m dispatch.

Note that an entry can be any of:
    - col, op   (for isnan, isnothing, ...)
    - col, op, val (for <, >, ...)
    - col, op, vals (for in, between, ....)
```

source

DataCurator.decode_function – Method.

```
decode_function(f::AbstractDict, glob::AbstractDict; condition=false)

Dispatched method for transform entries
```

source

DataCurator.delegate – Method.

```
delegate(config, template)
Uses the configuration, and template create by `create_template_from_toml', to execute the
↪  verifier as specified.
Returns the counters and file lists, if any are defined.
```

source

DataCurator.describe_image – Method.

```
describe_image(x::AbstractArray, axis::Int64)::DataFrame

Describe the array x sliced along axis.
```

source

DataCurator.dimg – Method.

```
dimg(x)

Describes the image argument as a collapsed Float64 array and returns the statistical moments.
```

source

DataCurator.dostep – Method.

```
doset(node, tuple, on_success)

Visitor function, evaluates if(tuple[1](node) -> tuple[2]

Dispatched by type for the condition-action or condition-action-counteraction
```

source

DataCurator.dostep – Method.

```
doset(node, tuple, on_success)

Visitor function, evaluates if(tuple[1](node) -> tuple[2] else -> tuple[3]
```

source

DataCurator.gaussian – Method.

```
gaussian(img, sigma)

Gaussian blur with σ
```

source

DataCurator.generate_counter – Function.

```
Make a count and counting functor that can be incremented by threads
```
c, ct = generate_counter()
ct(something)
@info c # "Counter = 1"
# Threaded version
pc, pct = generate_counter(true; x->reduce(*, size(x))))
a = zeros(3,3,3)
pct(a) # Threadsafe writes
# Printing the counter is not threadsafe, only read when all threads have finished.
@info pc # "Counter = 27"
```

source

DataCurator.getextent – Method.

```
For a bounding box, get the XY span (diagonal), Z range, and z center
```

source

DataCurator.handlecounters! – Method.

```
handlecounters!(entries, key, global_dict)

Decodes user specified counters (file size, count, etc)
Stores the result in global_dict
```

source

DataCurator.invert – Method.
```

```
invert(image)

For a normed ([0-1]) image, return  1 - image
```

source

DataCurator.laplacian – Method.

```
laplacian(image)

Laplacian of image (2nd derivative of intensity)
```

source

DataCurator.load_content – Method.

```
load_content(filename)

Tries to access common formats of content, currently supports tif/png/csv/txt
```

source

DataCurator.parse_acsym – Method.

```
Helper function to parse all functions
```

source

DataCurator.read_counter – Method.

```
read_counter(counter)

Sum a parallel or sequential counter where counter.data[threadid()]
```

source

DataCurator.reduce_image – Method.

```
reduce_image(image, operator, dimensions)

Apply a reduction (for example maximum projection) along the specified dimension, e.g. 3 for Z.
```

source

DataCurator.reduce_images – Method.

```
reduce_images(list, fname::AbstractString, op::AbstractString)

Given list of K-D images (tif), stack to K+1, then apply `op` along K+1 th dimension.
Save to fname in K-D tif

Example
maxproject = (list, fname) -> reduce_images(list, fname, "maximum")
```

source

DataCurator.send_to – Method.

```
/a/b/c, /a/b/c/d/e, /x/y
    if keeprelative
    -> /x/y/c/d/e
    if ~keeprelative
    -> /x/y/e
```

source

DataCurator.threshold_image – Method.

```
treshold(image, operator, value)

Set the image to zero where operator(image, value) == true.
Operator can be one of '<', '>', 'abs >', 'abs <'.
```

source

DataCurator.tmpcopy – Method.

```
tmpcopy(x; seed=0, length=40)

Create a temporary copy of file x (with same extension), of length 40.
40 means there's a 1/10000 for a collision of 2 identical files.
Note that this function is used on executing a template, in parallel, so it's not the total of
↪  number files being processed, but the number of files being processed in the same window of
↪  time.
Return the temporary file name
```

source

DataCurator.topdown – Method.

```
topdown(node, expander, visitor; context=nothing, inner=expand_sequential)
Recursively apply visitor onto node, until expander(node) -> []
If context is nothing, the visitor function gets the current node as sole arguments.
Otherwise, context is expected to contain: "node" => node, "level" => recursion level.
Inner is the delegate function that will execute the expand phase. Options are :
↪  expand_sequential, expand_threaded

Traversal is done in a pre-order way, e.g. visit before expanding.
```

source

DataCurator.transform_copy – Method.

```
transform_copy(x, f)
    x' = f(x) for a file or directory, a copy rather than a move. Refuses to act if x' exists.
```

source

DataCurator.transform_inplace – Method.

```
transform_inplace(x, f)
    x = f(x) for a file or directory. Refuses to act if x' exists.
```

source

DataCurator.verifier – Method.

```
verifier(node, templater::Dict, level::Int)
Dispatched function to verify at recursion level with conditions set in templater[level] for
↪  node.
Will apply templater[-1] as default if it's given, else no-op.
```

source

DataCurator.verifier – Method.

```
verifier(node, template::Vector, level::Int)
Dispatched function to verify at recursion level with conditions set in template for node.
Level is ignored for now, except to debug
```

source

DataCurator.verify_dispatch – Method.

```
verify_dispatch(context)
Use multiple dispatch to call the right function verifier.
```

source

DataCurator.verify_template – Method.

```
verify_template(start, template; expander=expand_filesystem, traversalpolicy=bottomup,
↪  parallel_policy="sequential")
Recursively verifies a dataset anchored at start using a given template.
For example, start can be the top directory of a filesystem.
A template has one of 2 forms:
    - template = [(condition, action_on_fail), (condition, action), ...]
        - where condition accepts a node and returns true if ok, false if not.
        - action is a function that accepts a node as argument, and is trigger when condition
        ↪  fails, example warn_on_fail logs a warning
Traversalpolicy is bottomup or topdown. For modifying actions bottomup is more stable.
Parallel_policy is one of "sequential" or "parallel". While parallel execution can be a lot
↪  faster, be very careful if your actions share global state.
```

source

DataCurator.wrap_transform – Method.

```
wrap_transform(x::AbstractString)

For file x, generate a temp copy before aggregation on a file list
```

source