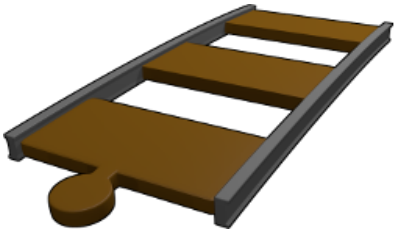


railroadtracks Documentation

Release 0.3.1

Laurent Gautier (laurent.gautier@novartis.com)

December 08, 2014

CONTENTS

A toolkit to connect (D|R)NA-Seq steps

CHAPTER

ONE

INTRODUCTION

> It is true that the most ancient peoples, the first librarians, employed a language quite different from the one we speak today; it is true that a few miles to the right, our language devolves into dialect and that ninety floors above, it becomes incomprehensible.
>
> —Jorge Luis Borges, The Library of Babel

The processing of Next-Generation Sequencing (NGS) data is generally achieved through a sequence of steps called a pipeline. Changing steps in the pipeline, or changing parameters for steps, or adding input data while wishing to compare the alternative outcomes requires tedious bookkeeping, tailored code for the alternatives, and possibly computing several times the tasks common to several alternatives.

`railroadtracks` is a Python toolkit with the following capabilities:

Track task dependencies The toolkit is keeping persistently the files used by all tasks in a project, and is providing intuitive ways to navigate complex dependency graphs.

Decouple declaration from execution *railroadtracks* allows the creation of all interconnected computational tasks required for a project before any computation is performed. This, added to the model aspect, allows consistency checks early instead of discovering issues after a number of computation tasks have already been performed. This is also permitting the use different scheduling systems within a project. The toolkit is providing an execution layer using `multiprocessing`[1], and custom execution layers can be added.

Implicit naming of result files The toolkit is making a clear separation between the meta-data for a result file (how was a result file obtained) from its file name. User can write complete pipelines, while only specifying the file names for the input data at the root of the dependency graph.

Incremental writing of analysis pipelines The persistent tracking of task dependencies permits the incremental writing of data processing pipelines without explicit bookkeeping nor unnecessarily repeated computations. Pipelines can be modified, with new input files added, tools used changed, while only new required computations are performed.

Standardization into models By providing a system to wrap tools into model classes, `railroadtracks` allows a standardization of these tools that allows both an easy connection between steps and the ability to swap between similar tools. This and the implicit naming of result files makes the combination of tools or parameters trivial.

REPL-aware `railroadtracks` was designed with interactive programming (REPL) in mind, and ipython and the ipython notebook as a showcase environment. High-level rendering of objects is provided (text, HTML) as well as visualization (provenance and destination graphs) are provided. The autocompletion of Python namespaces is also part of our design.

Models DNA and RNA sequencing (batteries included) While it is possible to extend `railroadtracks` with custom models, a number of modeled steps for generating synthetic reads, aligning, quantifying, testing for differential expression, are already included.

[1] http://docs.python.org/library/multiprocessing.html#module-multiprocessing

1.1 Tutorial

Tutorial as an IPython notebook [html | pynb]

CHAPTER

TWO

INSTALLATION

2.1 Requirements

This was developed for Python 2.7.

Note: While a keen attention has been paid to compatibility with Python 3, the use of metaclasses in this packages and the incompatibility of syntax with them between Python 2 and 3 prevents this package from currently working with Python 3. This is the major blocker for Python 3-compatibility and we plan on using `six` to solve this.

A number of python packages are required or recommended (all of which are available on *Pypi <https://pypi.python.org/pypi>*):

- *jinja2 <http://jinja.pocoo.org/>*
- *networkx <https://networkx.github.io/>*
- *pygraphviz <https://pygraphviz.github.io/>*
- *enum34 <https://pypi.python.org/pypi/enum34>*
- *ngs_plumbing <http://pythonhosted.org/ngs_plumbing/>*

The model steps included require a number of third-party tools in order to have all steps working. A missing tool does not prevent this package from working, but the related functionality will not be working.

At the time of writing, the tools are:

- *samtools <http://samtools.sourceforge.net>*
- *bowtie <http://bowtie-bio.sourceforge.net/>*
- *bowtie2 <http://bowtie-bio.sourceforge.net/bowtie2>*
- *tophat <http://ccb.jhu.edu/software/tophat/index.shtml>*
- *tophat2 <http://ccb.jhu.edu/software/tophat/index.shtml>*
- *GSNAP <http://research-pub.gene.com/gmap/>*
- *sailfish <http://www.cs.cmu.edu/~ckingsf/software/sailfish/>*
- *STAR <https://github.com/alexdobin/STAR>*
- *HTSeq <http://www-huber.embl.de/users/anders/HTSeq>* (for the executable *htseq-count*)
- ***R <http://www.r-project.org>*, with libraries:**
  - *rjson <http://cran.r-project.org/package=rjson>* (required for any interaction with R)

  and bioconductor libraries

- *Rsubread* <*http://www.bioconductor.org/packages/release/bioc/html/Rsubread.html*> (for *featureCount*)
- *DESeq* <*http://www.bioconductor.org/packages/release/bioc/html/DESeq.html*>
- *DESeq2* <*http://www.bioconductor.org/packages/release/bioc/html/DESeq2.html*>
- *limma* <*http://www.bioconductor.org/packages/release/bioc/html/limma.html*> (for the method *voom*)
- *edgeR* <*http://www.bioconductor.org/packages/release/bioc/html/edgeR.html*>

Note: Running the tests with *-v* (see below) will tell which tools are missing to have all functionality.

The package is including a small genome (Enterobacteria phage MS2 isolate ST4 retrieved from the NCBI site: http://www.ncbi.nlm.nih.gov/nuccore/EF204940.1 - bibliographic reference *Friedman SD et al., J Virol. 2009 Nov;83(21):11233-43. doi: 10.1128/JVI.01308-09.* <*http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2772794/*>) as FASTA, as well as GTF/GFF derived from its genome annotation, for the purpose of running tests and examples.

2.2 Installation

This is a regular Python package. It can be installed with

```
python setup.py install
```

Running the unit-tests can be done after installation

```
python -m railroadtracks.tests # for a detail of the tests, add "-v"
```

Note: While the unit tests use a very small dataset included in the package (a viral genome), running the test can take a bit of disk space (for instance, at the time of writing STAR indexes take a rather large amount of space for an otherwise rather small genome).

CHAPTER

THREE

RECIPES

The model, together with the persistence layer, is designed to make the writing of sequences of steps for RNA-Seq data processing rather simple and and make changing to an alternative step

(e.g., aligner, differential expression method, etc...)

trivial. This is also designed to make coexisting variants something a user does not have to worry about (unless inclined to - the system is open).

Note: The general principles to remember are limited to:

- Steps require assets to run (and optionally parameters)
- Assets are constituted of two groups: *source* and *target* elements
- Parameters are optionally given

The bundle of source and target assets, parameters, and a step represents a *task*. Explanations about step and assets follow.

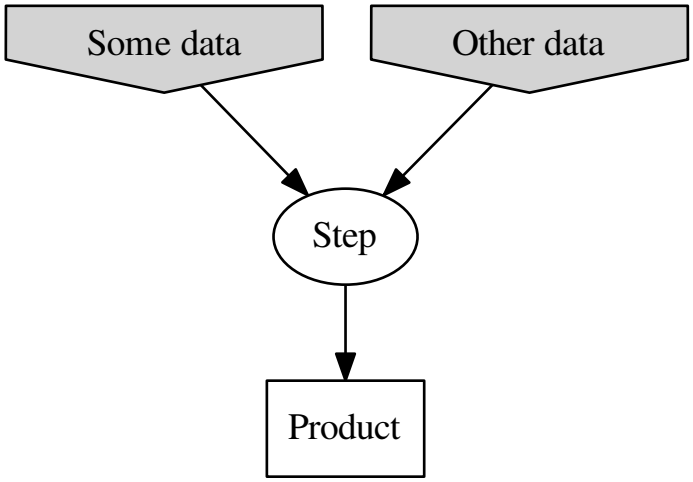
```
>>> source = SomeStep.Assets.Source( inputentity )
>>> target = SomeStep.Assets.Target( outputentity )
>>> assets = SomeStep.Assets(source, target)
>>> step = SomeStep( pathtoexecutable )
>>> step.run( assets )
```

3.1 Steps and assets

3.1.1 Concept

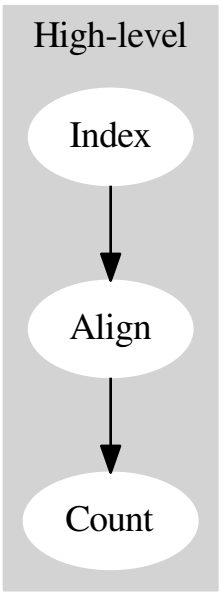
All steps in the process are connected through intermediate data files which we call assets. Bioinformatics tools are almost always designed to operate on files, with the occasional pipes being used.

A *step* can be represented as the step itself with input files (the *source* assets) and output files (the *target* assets).

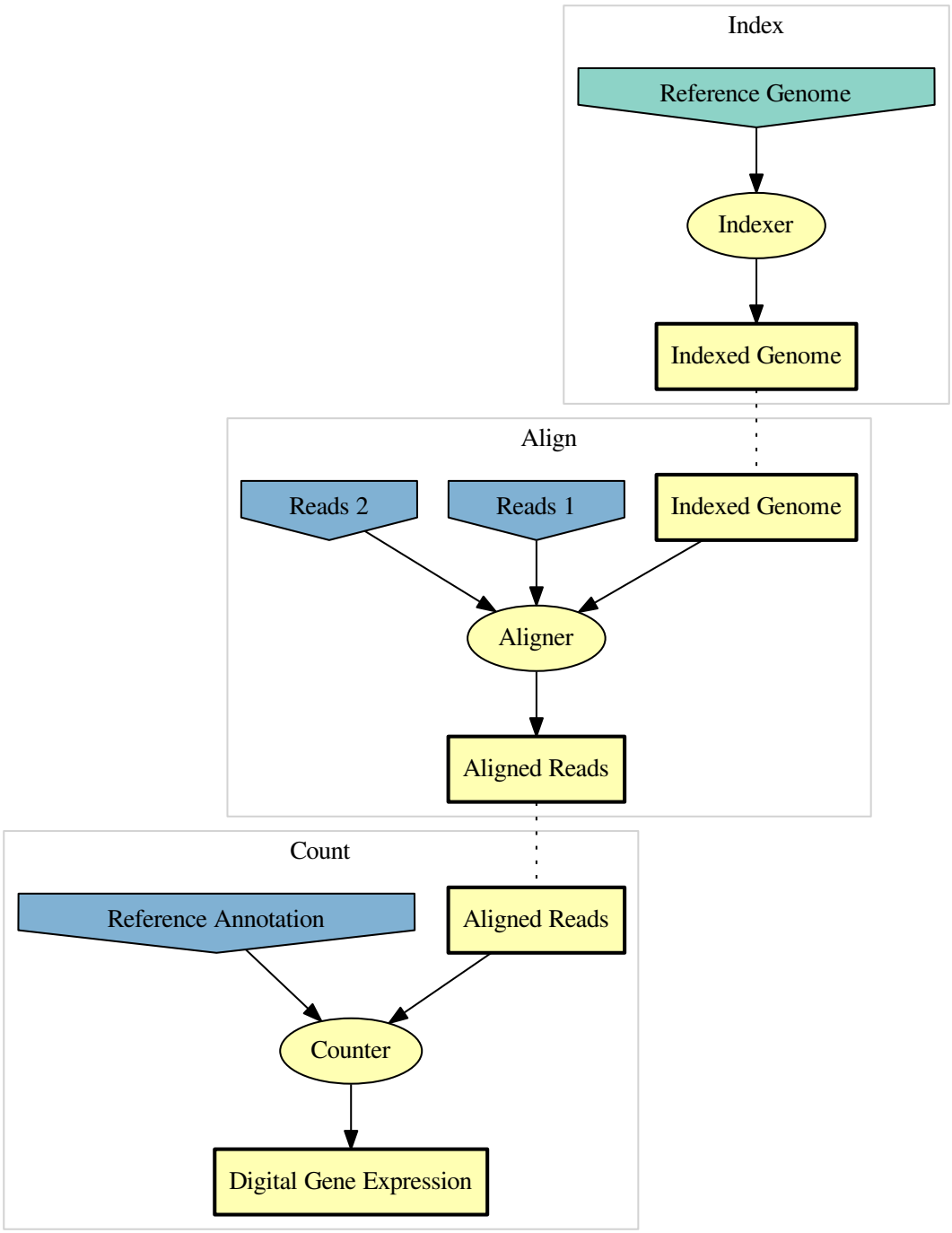


This group of nodes (here 4 nodes: 2 source, 1 step, 1 target) can also be collapsed as a *step*, and the representation of a workflow be the graph connecting this summary representation.

For example, the initial steps for aligner-based RNA-Seq can be represented as:



The Step-and-Assets graph will then look like this:



The nodes *Indexed Genome* and *Aligned Reads* are duplicated for clarity with the grouping of nodes, but it is the same entity saved on disk produced and used by the two steps.

When using the framework, the easiest way to think about it is to start from a step (child class of `StepAbstract`),

and look for the attribute `Assets`. That attribute is a class representing the step's assets, and is itself containing 2 class attributes `Source` and `Target` (themselves classes as well).

For example with the class modeling the aligner "STAR":

```
import railroadtracks.rnaseq import StarIndex, StarAlign

# assets for the indexing of references (indexed for the alignment)
StarIndex.Assets
# assets for aligning reads against indexed references
StarAlign.Assets
```

The assets are divided into two sets, the *Source* and the *Target*, for the files the step is using and the files the step is producing respectively. `Source` and `Target` are themselves classes.

```
import railroadtracks.rnaseq import StarIndex, StarAlign

# assets for the indexing of references (indexed for the alignment)
fastaref = rnaseq.SavedFASTA('/path/to/referencegenome.fasta')
indexedref = rnaseq.FilePattern('/path/to/indexprefix')
StarIndex.Assets(StarIndex.Assets.Source(fastaref),
                 StarIndex.Assets.Target(indexedref))
```

The results are somewhat verbose, but if an IDE or an advanced interactive shell such as *ipython* is used, autocompletion will make writing such statements relatively painless, and mostly intuitive. One can just start from the step considered (`StarIndex` in the example above) and everything can be derived from it.

3.1.2 Unspecified assets

In a traditional approach, for example a sequence of commands in *bash*, the name of files must be specified at each step.

We are proposing an alternative with which a full *recipe* can be written without having to take care of file names for derived data (which can be a relative burden when considering to process the same data in many alternative ways). Only the original data files such as the reference genome, or the experimental data from the samples and sequencing, are specified and the other file names will be generated automatically.

Objects inheriting from `AssetSet` are expected to have a method `AssetSet.createundefined()` that creates an undefined set of assets, and this can be used in recipes (see example below).

```
Foo = core.assetfactory('Foo', [core.AssetAttr('bar', core.File, '')])
# create an undefined set of assets of type Foo
undefoo = Foo.createundefined()
```

3.1.3 Notes about the design

Data analysis is often a REPL activity, and we are keeping this in mind. The writing of a recipes tries to provide:

- autocompletion-based discovery. Documentation is rarely read back-to-back (congratulations for making it this far though), and dynamic scientists often proceed by trial-and-error with software. The package is trying to provide benevolent support when doing so.
- fail early whenever possible (that is before long computations have already been performed)
- allow the writing of a full sequence of steps and run them unattended (tasks to be performed are stored, and executed when the user wants to)

3.2 Setup

```
# -- initialization boiler plate code
import tempfile
from railroadtracks import hortator, rnaseq, easy
from environment import Executable

wd = tempfile.mkdtemp()
project = easy.Project(rnaseq, wd=wd)

# declare the 3rd-party command-line tools we will use
env = easy.Environment(rnaseq)
```

The package is also able to generate a small dataset based on a phage:

```
# Phage genome shipped with the package for testing purposes
import railroadtracks.model.simulate
PHAGEFASTA = railroadtracks.model.simulate.PHAGEFASTA
PHAGEGFF = railroadtracks.model.simulate.PHAGEGFF

# create random data for 6 samples (just testing here)
nsamples = 6
samplereads = list()
with open(PHAGEFASTA) as fasta_fh:
    reference = next(railroadtracks.model.simulate.readfasta_iter(fasta_fh))
for sample_i in range(nsamples):
    read1_fh = tempfile.NamedTemporaryFile(prefix='read1', suffix='.fq')
    read2_fh = tempfile.NamedTemporaryFile(prefix='read2', suffix='.fq')
    read1_fh, read2_fh = railroadtracks.model.simulate.randomPEreads(read1_fh,
                                                                     read2_fh,
                                                                     reference)
    samplereads.append((read1_fh, read2_fh))

sampleinfo_fh = tempfile.NamedTemporaryFile(suffix='.csv')
csv_w = csv.writer(sampleinfo_fh)
csv_w.writerow(['sample_id', 'group'])
for i in range(6):
    csv_w.writerow([str(i), ('A','B')[i%2]])
sampleinfo_fh.flush()
referenceannotation = rnaseq.SavedGFF(PHAGEGFF)
```

3.3 Simple recipe

```
# steps used
bowtie2index = env.activities.INDEX.bowtie2build
bowtie2align = env.activities.ALIGN.bowtie2
htseqcount = env.activities.QUANTIFY.htseqcount
merge = env.activities.UTILITY.columnmerger
edger = env.activities.DIFFEXP.edger

import easy

# sequence of tasks to run
torun = list()

# index for alignment
```

```
Assets = bowtie2index.Assets
assets = Assets(Assets.Source(rnaseq.SavedFASTA(reference_fn)),
                Assets.Target.createundefined())
task_index = project.add_task(bowtie2index, assets)
torun.append(task_index)

# process all samples
sample_counts = list()
for read1_fh, read2_fh in samplereads:
    # align
    Assets = bowtie2align.Assets
    assets = Assets(Assets.Source(task_index.call.assets.target.indexfilepattern,
                                  rnaseq.FASTQPossiblyGzipCompressed(read1_fh.name),
                                  rnaseq.FASTQPossiblyGzipCompressed(read2_fh.name)),
                    Assets.Target.createundefined())
    task_align = project.add_task(bowtie2align, assets)
    torun.append(task_align)

    # quantify
    # (non-default parameters to fit our demo GFF)
    params = rnaseq.HTSeqCount._noexons_parameters
    Assets = htseqcount.Assets
    assets = Assets(Assets.Source(task_align.call.assets.target.alignment,
                                  rnaseq.SavedGFF(referenceannotation)),
                    Assets.Target.createundefined())
    task_quantify = project.add_task(htseqcount,
                                     assets,
                                     parameters=params)
    torun.append(task_quantify)
    # keep a pointer to the counts,
    # as we will use them in the merge step
    sample_counts.append(task_quantify.call.assets)

# merge the sample data into a table
# (so differential expression can be computed)
Assets = merge.Assets
counts = tuple(x.target.counts for x in sample_counts)
assets = Assets(Assets.Source(rnaseq.SavedCSVSequence(counts)),
                merge.Assets.Target.createundefined())
task_merge = project.add_task(merge,
                              assets,
                              parameters=("0","1"))
torun.append(task_merge)

# differential expression with edgeR
Assets = edger.Assets
assets = Assets(Assets.Source(task_merge.call.assets.target.counts,
                              rnaseq.SavedCSV(sampleinfo_fh.name)),
                Assets.Target.createundefined())
task_de = project.add_task(edger,
                           assets)

# run the tasks
for task in torun:
    # run only if not done
    if task.info[1] != hortator._TASK_DONE:
        task.execute()
```

```
# get results
final_storedentities = project.get_targetsofactivity(rnaseq.ACTIVITY.DIFFEXP)

# get the step that created the results files
final_steps = list()
for stored_entity in final_storedentities:
    final_steps.append(project.todo._cache.get_parenttask_of_storedentity(stored_entity))
```

The variable *wd* contains the directory with all intermediate data and the final results, and *db_dn* is the database file.

Note: If you clean up after yourself, but want to run the next recipe in this documentation, the setup step will have to be run again.

3.4 Loops, nested loops, and many variants

```
import easy

torun = list()

# bowtie
bowtie1index = env.activities.INDEX.bowtiebuild
bowtie1align = env.activities.ALIGN.bowtie
Assets = bowtie1index.Assets
fa_file = rnaseq.SavedFASTA(reference_fn)
task_index_bowtie1 = project.add_task(bowtie1index,
                                      Assets(Assets.Source(fa_file),
                                             None))
torun.append(task_index_bowtie1)

# bowtie2
bowtie2index = env.activities.INDEX.bowtie2build
bowtie2align = env.activities.ALIGN.bowtie2
Assets = bowtie2index.Assets
fa_file = rnaseq.SavedFASTA(reference_fn)
task_index_bowtie2 = project.add_task(bowtie2index,
                                      Assets(Assets.Source(fa_file),
                                             None))
torun.append(task_index_bowtie2)

# STAR
starindex = env.activities.INDEX.starindex
staralign = env.activities.ALIGN.staralign
Assets = starindex.Assets
fa_file = rnaseq.SavedFASTA(reference_fn)
task_index_star = project.add_task(starindex,
                                   Assets(Assets.Source(fa_file),
                                          None))
torun.append(task_index_star)

# TopHat2
# (index from bowtie2 used)
#tophat2 = env.activities.ALIGN.tophat2

# HTSeqCount
htseqcount = env.activities.QUANTIFY.featurecount
```

```
# Merge columns (obtained from counting)
merge = env.activities.UTILITY.columnmerger

# EdgeR, DESeq, DESeq2, and LIMMA voom
edger = env.activities.DIFFEXP.edger
deseq = env.activities.DIFFEXP.deseq
deseq2 = env.activities.DIFFEXP.deseq2
voom = env.activities.DIFFEXP.limmavoom


# Now explore the different alignment presets in bowtie2, and vanilla star
from itertools import cycle
from collections import namedtuple
Options = namedtuple('Options', 'aligner assets_index parameters')
# Try various presets for bowtie2
bowtie2_parameters = (('--very-fast', ), ('--fast', ),
                      ('--sensitive', ), ('--very-sensitive', ))
options = [Options(*x) for x in zip(cycle((bowtie2align,)),
                                    cycle((task_index_bowtie2.call.assets.target,)),
                                    bowtie2_parameters)]

# add bowtie
options.append(Options(bowtie1align, task_index_bowtie1.call.assets.target, tuple()))
# add STAR (vanilla, no specific options beside the size of index k-mers)
options.append(Options(staralign,
                       task_index_star.call.assets.target,
                       ('--genomeChrBinNbits', '12')))
# add TopHat2
#options.append(Options(tophat2, task_index_bowtie2.call.assets.target, tuple()))

# loop over the options
for option in options:
    sample_counts = list()
    # loop over the samples
    for sample_i in range(nsamples):
        read1_fh, read2_fh = samplereads[sample_i]
        # align
        Assets = option.aligner.Assets
        assets = Assets(Assets.Source(option.assets_index.indexfilepattern,
                                      rnaseq.FASTQPossiblyGzipCompressed(read1_fh.name),
                                      rnaseq.FASTQPossiblyGzipCompressed(read2_fh.name)),
                        Assets.Target.createundefined())
        task_align = project.add_task(option.aligner,
                                      assets,
                                      parameters=option.parameters)
        torun.append(task_align)

        # quantify
        # (non-default parameters to fit our demo GFF)
        Assets = htseqcount.Assets
        assets = Assets(Assets.Source(task_align.call.assets.target.alignment,
                                      rnaseq.SavedGFF(referenceannotation)),
                        Assets.Target.createundefined())
        task_quantify = project.add_task(htseqcount,
                                         assets)
        torun.append(task_quantify)

        # keep a pointer to the counts, as we will use it in the merge step
```

```
            sample_counts.append(task_quantify.call.assets)

        # merge the sample data into a table (so differential expression can be computed)
        Assets = merge.Assets
        source = Assets.Source(rnaseq.SavedCSVSequence(tuple(x.target.counts\
                                                             for x in sample_counts)))
        assets_merge = Assets(source,
                              Assets.Target.createundefined())
        task_merge = project.add_task(merge,
                                      assets_merge,
                                      parameters=("0","1"))
        torun.append(task_merge)

        # differential expression with edgeR, deseq2, and voom
        # (deseq is too whimsical for tests)
        for diffexp in (edger, deseq, deseq2, voom):
            Assets = diffexp.Assets
            assets = Assets(Assets.Source(task_merge.call.assets.target.counts,
                                          core.File(sampleinfo_fh.name)),
                            Assets.Target.createundefined())
            task_de = project.add_task(diffexp,assets)
            torun.append(task_de)

    # run the tasks
    for task in torun:
        if task.info[1] != hortator._TASK_DONE:
            try:
                task.execute()
                status = easy.hortator._TASK_DONE
            except:
                status = easy.hortator._TASK_FAILED
        project.todo._cache.step_concrete_state(task.task_id,
                                                easy.hortator._TASK_STATUS_LIST[status])
```

3.5 Docstrings

Module aiming at making common operations "easy".

class `railroadtracks.easy.`**`ActivityCount`**
ActivityCount(count, name, status)

`count`
Alias for field number 0

`name`
Alias for field number 1

`status`
Alias for field number 2

class `railroadtracks.easy.`**`Asset`**(*project*, *savedentity*, *asset_id*)
An asset is either used by a task (then it is a source asset) or is produced by a task (then it is a target asset).

`entity`
Instance of `SavedEntityAbstract` (or rather a child class thereof).

class `railroadtracks.easy.`**`DbID`**
DbID(id,)

id
Alias for field number 0

class railroadtracks.easy.**Environment**(*model*)
Represent the current environment in a (presumably) easy way for writing recipes.

__init__(*model*)

Parameters **model** – Python module following the railroadtracks model.

activities
Access the activities declared by the model as attributes.

stepclasses
Steps.

stepinstances
Default instance for the steps (created from the default executables in the PATH).

class railroadtracks.easy.**Project**(*model*, *wd='railroadtracks_project'*, *db_fn=None*, *force_create=False*)
A project, that is a directory containing data as well as a persistant storage of steps and how derived data and final results were obtained.

__init__(*model*, *wd='railroadtracks_project'*, *db_fn=None*, *force_create=False*)

Parameters

- **wd** (str[1]) – Name of a working directory (where all intermediate and final results will be saved.
- **db_fn** (str[2] or None) – Name of a file name for the database. If None, use the file "railroadtracks.db" in the directory specified in "wd".

Return type a tuple with (hortator.StepGraph, working directory as a str[3], file name for the database and a str[4])

add_task(*step*, *assets*, *parameters=()*, *tag=1*)
Add a task to the project. If any of the assets' targets is not defined, it will be defined automatically.

Parameters

- **step** –
- **assets** –
- **parameters** –
- **tag** – a tag (to differentiate repetitions of the exact same task)

db_fn
Path to the database file

get_targetsofactivity(*activity*)
Retrieve the targets of steps performing a specific activity. (calls the method of the same name in the contained PersistentTaskList) :param activity: an activity :type activity: Enum

get_targetsoftype(*obj*)
Retrieve the targets having a given type. (calls the method of the same name in the contained

[1] http://docs.python.org/library/functions.html#str
[2] http://docs.python.org/library/functions.html#str
[3] http://docs.python.org/library/functions.html#str
[4] http://docs.python.org/library/functions.html#str

PersistentTaskList) :param obj: a type, an instance, or a type name :type obj: type[5], Object, or str[6]

get_task(*task_id*)
Given a task ID, retrieve the associated task.

iter_srcassets(*task*)
Return the source files for a given task (calls the method of the same name in the contained PersistentTaskList) :param task: a Task.

iter_targetassets(*task*)
Return the target files for a given task (calls the method of the same name in the contained PersistentTaskList) :param task: a Task.

model
Model used in the project.

newproject
Tell whether this is a new project, rather than a existing project (re)opened.

wd
Working directory. The directory in which derived data is stored.

class railroadtracks.easy.**Task**(*project*, *call*, *task_id*)
A task.

all_child_tasks()
Get all the child tasks, direct or indirect.

call
A railroadtracks.unifex.Call object.

child_tasks()
Tasks depending on the current task.

dirname
Directory in which the targets of the task are saved.

execute()
Execute the task. Note that the status of the task known to the project is not updated.

info
Status information for the task.

parent_tasks()
Tasks the current task is depending on.

primordial_tasks()
Direct or indirect parent tasks with source assets that do not have parent tasks themselves. In other words, root nodes in the dependency graph connected to this task.

This method is useful to identify the raw data results are derived from.

project
Project in which the task is defined.

status
Get/Set the status of the task in the project. The value must be a valid task status.

task_id
ID of the task in the project.

[5] http://docs.python.org/library/functions.html#type
[6] http://docs.python.org/library/functions.html#str

unifex_cmd()
Parsed command to run the task.

To make a string to copy/paste into a shell script, use `subprocess.list2cmdline()`.

unifex_cmdline()
Command line as could be run in a shell. (This is a wrapper around subprocess.list2cmdline(self.unifex_cmd()))

railroadtracks.easy.**Task2**
alias of `Step`

class railroadtracks.easy.**TaskSet**(*project=None*, *iterable=()*)
Ordered set of tasks. This class can be used for independent tasks for which the order of completion does not matter, and provide a container for tasks that can run in parallel.

add(*task*)
Add a task.

remove(*task*)
remove a task.

status()
Return a *Counter* of status labels.

values()
Return the tasks as a tuple (Python 2) or an iterator (Python 3).

railroadtracks.easy.**call_factory**(*project*, *step_concrete_id*, *stepobj*, *assets*, *parameters=()*)

Parameters

- **stepobj** (instance of `StepAbstract`) – Step object
- **assets** (instance of `AssetsStep`) – assets

Return type `function`

railroadtracks.easy.**command_line**(*project*, *stepconcrete_id*, *stepobj*, *assets*, *parameters=()*)

Parameters

- **stepobj** (instance of `StepAbstract`) – Step object
- **assets** (instance of `AssetsStep`) – assets

Return type a named `tuple` with the arguments for a command line call

3.6 Troubleshooting

The standard Python module `logging`[7] is used for logging, and its documentation should be checked. For example, a very simple way to activate logging at the level *DEBUG* on *stdout*:

```
import railroadtracks
logger = railroadtracks.logger
logger.setLevel(railroadtracks.logging.DEBUG)
railroadtracks.basicConfig('/tmp/stdout')
```

[7] http://docs.python.org/library/logging.html#module-logging

CHAPTER

FOUR

MODEL

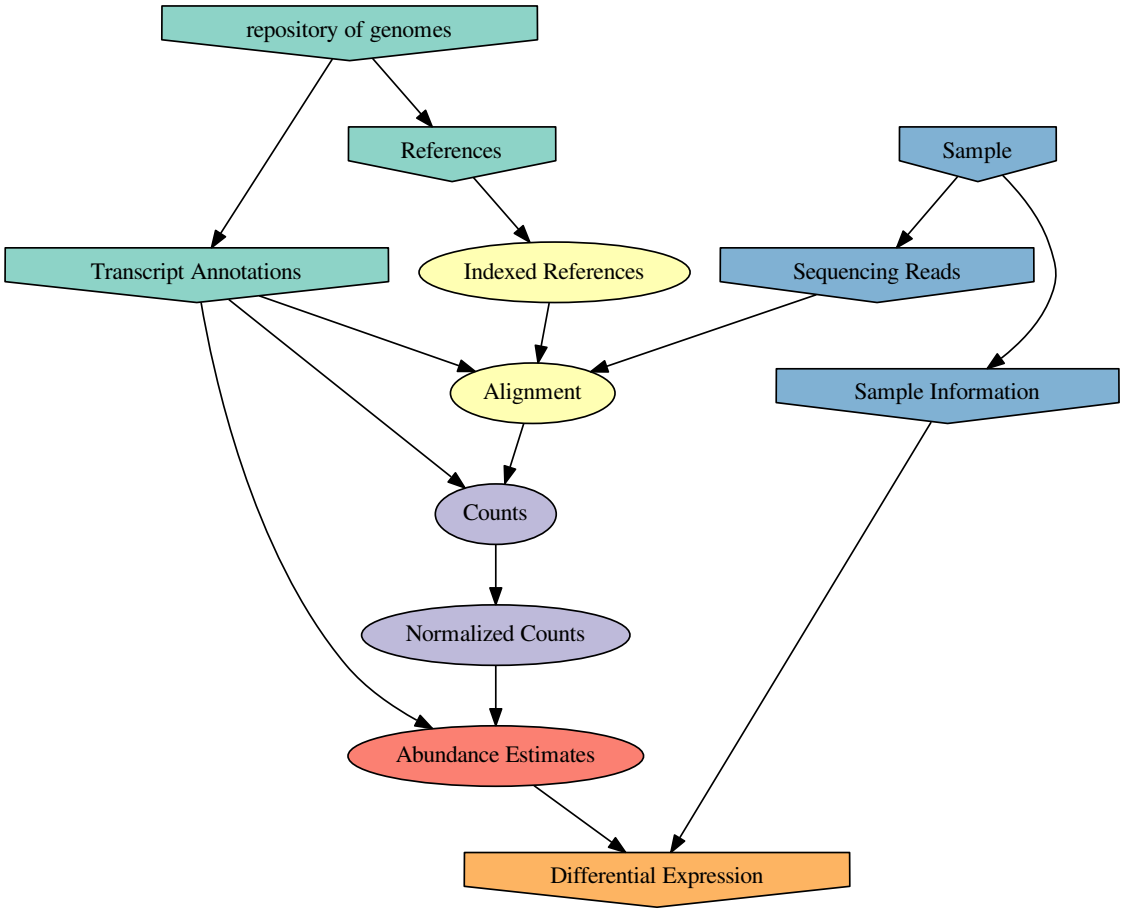
4.1 Overview

The steps in a RNA-Seq sequence of operations are captured in a model. The purpose is to formalize the steps enough to simplify the use of various tools for each step, yet permit enough flexibility to allow an easy integration of additional tools and approaches.

A canonical graph of steps for RNA-Seq is shown below.

Note: There is no enforcement that a sequence of steps adheres strictly to this. The model is also defining the notion of *activities*, and it is possible that one step performs several activities (this will be the case for an in-house monolithic pipeline, for example). This feature allows us to integrate such steps into the comparison.

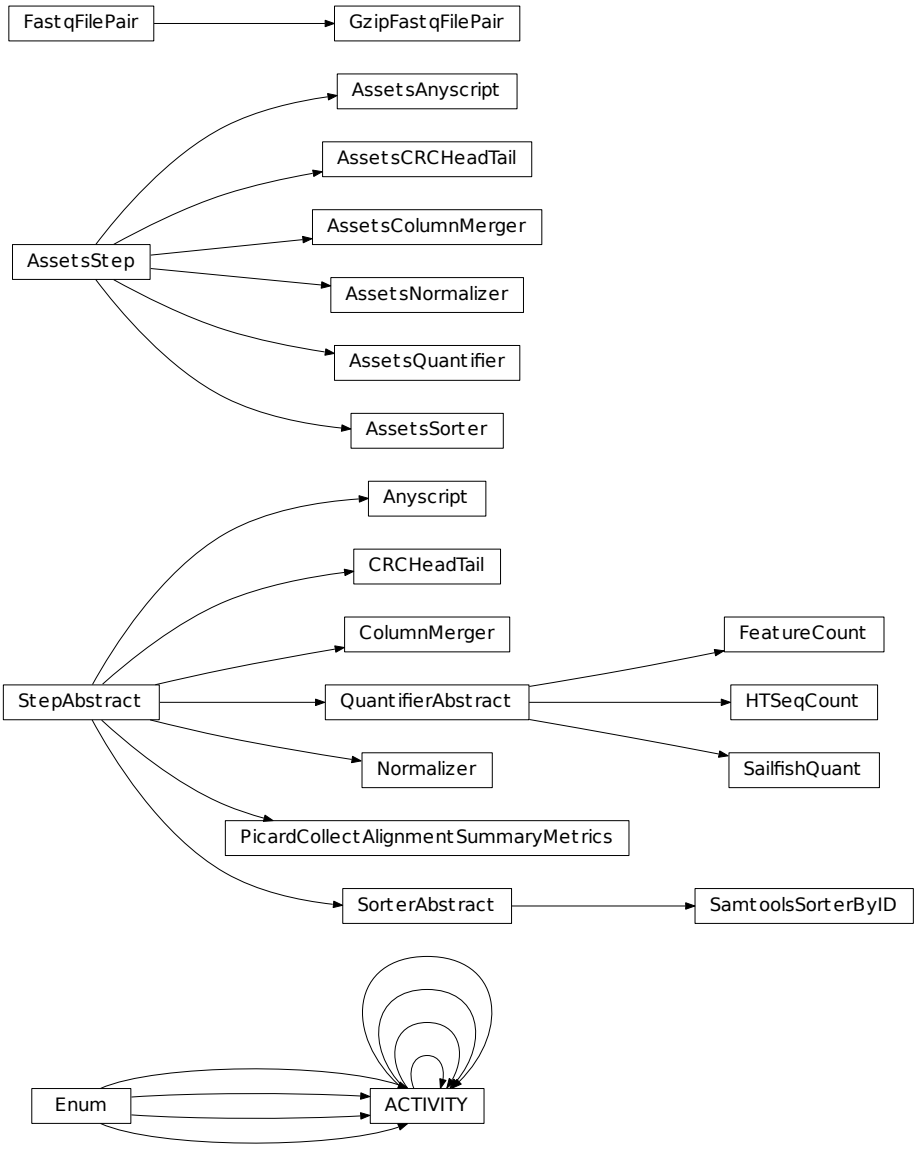
The model is directly used to provide an unified execution scheme (see *Unified execution*), and constitutes the basis for writing "recipes" (see *Recipes*).



4.2 Activities

Steps can perform one or several activities. The list of possible activities is in `ACTIVITY`

4.3 Inheritance diagram



4.4 Docstrings

Simple model for the expression analysis using RNA-Seq

class `railroadtracks.rnaseq.`**`ACTIVITY`**

Activities that can be performed by the different steps modeled. (note: steps can combine several activities - the most obvious example is existing monolithic pipelines)

class railroadtracks.rnaseq.**Anyscript**(*executable=None*)
Do anything.

Assets
alias of AssetsAnyscript

class railroadtracks.rnaseq.**AssetsAnyscript**(*source*, *target=None*)
Assets for Anyscript

class railroadtracks.rnaseq.**AssetsCRCHeadTail**(*source*, *target=None*)
Assets for CRCHeadTail

class railroadtracks.rnaseq.**CRCHeadTail**(*executable=None*)
Compute a CRC32-based checksum on the beginning (head) and end (tail) of a file as a cheap way to check whether 2 files contain identical data. Might be useful with large files, however its main purpose is testing.

Assets
alias of AssetsCRCHeadTail

class railroadtracks.rnaseq.**GzipFastqFilePair**(*filename1*, *filename2*, ***kwargs*)
Pair of FASTQ files The default iterator will go through the paired entries in the file (not the rows), assuming that they are in the same order. No check that this is the case (using IDs) is performed.

iter_seqqual_pairs()
Iterate over the pairs of (sequence+quality) in the file.

class railroadtracks.rnaseq.**SamtoolsSorterByID**(*executable=None*)

class railroadtracks.rnaseq.**SorterAbstract**
A sorting step.

CHAPTER

FIVE

EXTENDING THE FRAMEWORK

5.1 Writing custom steps

5.1.1 Simple case

A lot of the boilerplate handling is handled by parent classes. A custom class for a new tool will have to implement code to:

- create an instance (constructor)
- get the version
- run the step

The base class for steps is the abstract class `rnaseq.AssetStep`:

```
class StepAbstract(object):
    """ Abstract parent for steps. """
    __metaclass__ = abc.ABCMeta
    # monicker under which the step will be known (must be unique within a step list)
    _name = abc.abstractproperty()
    # default name for executable associated with the class
    _default_execpath = abc.abstractproperty()

    # class of assets for the step (must be a child of :class:`AssetsStep`)
    Assets = abc.abstractproperty()

    activities = abc.abstractproperty(None, None, None,
                                      "Activities associated with the step.")

    version = abc.abstractproperty(None, None, None,
                                   "Version of the executable associated with the step.")

    @abc.abstractmethod
    def run(self, assets, parameters=tuple()):
        """
        :param assets: Assets to run the step with
        :type assets: :class:`AssetsStep`
        :param parameters: optional parameters
        """
        raise NotImplementedError(_NOTIMPLEMENTED_ABSTRACT)

    def uei(self, assets, parameters = tuple()):
        # FIXME: should return the unified execution command line
        uei = UnifiedExecInfo(self._execpath, self._name,
```

```
                                assets.source, assets.target, parameters,
                                None, None) #logging_file and logging_level
        return uei
```

Custom steps will just have to implement the methods and properties. For example, counting the reads after alignment with htseq-count is defined as a child class of `rnaseq.QuantifierAbstract`, where `Assets` are defined:

```
class QuantifierAbstract(core.StepAbstract):
    __metaclass__ = ABCMeta
    Assets = AssetsQuantifier
    activities = (ACTIVITY.QUANTIFY, )
```

Note: There is a class for each activity defined (see *Activities*), and these can be used as parent class for new steps performing one activity The `Assets` is inherited from its parent class, and does not need to be specified further.

```
class AssetsQuantifier(core.AssetsStep):
    Source = core.assetfactory('Source', [core.AssetAttr('alignedreads', BAMFile, ''),
                                          core.AssetAttr('annotationfile', SavedGFF, '')])
    Target = core.assetfactory('Target', [core.AssetAttr('counts', SavedCSV, '')])
```

One can note that specific subclasses of `core.SavedEntityAbstract` can be specified (here `SavedCSV` to indicate that file produced by the counting step is a CSV file). The definition of assets represents a way to add a static typing flavor to Python, as a number of checks are performed behind the hood by `railroadtracks.rnaseq`.

The definition of `rnaseq.HTSeqCount` is then implementing the remaining methods and attributes

```
class HTSeqCount(QuantifierAbstract):

    _name = 'htseqcount'
    _default_execpath = 'htseq-count'
    _separator = '\t'
    # set of parameters to get htseq-count to work with references such
    # as bacteria or viruses (stored as a class attribute for convenience)
    _noexons_parameters = ('--type=CDS', '--idattr=db_xref', '--stranded=no')

    def __init__(self, executable=None):
        if executable is None:
            executable = type(self)._default_execpath
        self._execpath = executable
        self._version = None

    @property
    def version(self):
        if self._version is None:
            cmd = [self._execpath, '-h']
            try:
                logger.debug(subprocess.list2cmdline(cmd))
                res = subprocess.check_output(cmd)
            except OSError as ose:
                raise UnifexError("""Command: %s
                %s""" % (' '.join(cmd), ose))

            m = re.match('^.*( version )?([^ \n]+)\.$', res.split('\n')[-2])
            if m is None:
                raise RuntimeError('Could not find the version number.')
            self._version = m.groups()[1]
        return self._version
```

```
    def run(self, assets, parameters = tuple()):
        # FIXME: shouldn't strandedness be a better part of the model ?
        source = assets.source
        sortedbam = source.alignedreads
        if not isinstance(source.alignedreads, BAMFileSortedByID):
            # htseq-count requires sorted entries
            warnings.warn(("The source asset '%s' should ideally be sorted by read IDs. " +\
                           "We are sorting the file; use explicitly a '%s' rather than a '%s' "+\
                           "for better performances, as well as for reproducibility issues "+\
                           "(the sorting will use whatever 'samtools` is first found in the PATH)") \
                          % ("alignedreads", BAMFileSortedByID.__name__, BAMFile.__name__))
            output_dir = os.path.dirname(assets.target.counts.name)
            # temp file name for the sorted output
            sortedbam_fh = tempfile.NamedTemporaryFile(dir=output_dir, suffix=".bam", delete=False)
            # (cleaning temp files handled by Python, except sortedsam)
            # -- sort
            sorter = SamtoolsSorterByID()
            sorter_assets = sorter.Assets(sorter.Assets.Source(source.alignedreads),
                                          sorter.Assets.Target(BAMFile(sortedbam_fh.name)))
            sorter.run(sorter_assets)
            # sanity check:
            if os.stat(sorter_assets.target.sortedbam.name).st_size == 0:
                warnings.warn('The sorted BAM file is empty.')
            sortedbam = sorter_assets.target.sortedbam
        else:
            sortedbam_fh = None

        # BAM to SAM
        cmd_bam2sam = ['samtools', 'view', sortedbam.name]

        # build command line
        cmd_count = [self._execpath, ]
        cmd_count.extend(parameters)
        cmd_count.extend(['-', source.annotationfile.name])
        cmd = cmd_bam2sam + ['|', ] + cmd_count

        logger.debug(subprocess.list2cmdline(cmd))
        with open(os.devnull, "w") as fnull, \
             open(assets.target.counts.name, 'w') as output_fh:
            csv_w = csv.writer(output_fh)
            # HTSeq-count does not use column names in its output, unfortunately,
            # so we correct that
            csv_w.writerow(['ID','count'])
            p_bam2sam = subprocess.Popen(cmd_bam2sam, stdout=subprocess.PIPE, stderr=fnull)
            p_htseq = subprocess.Popen(cmd_count,
                                       stdin = p_bam2sam.stdout,
                                       stdout = subprocess.PIPE,
                                       stderr = fnull)
            csv_r = csv.reader(p_htseq.stdout, delimiter='\t')
            for row in csv_r:
                csv_w.writerow(row)
                p_htseq.stdout.flush()
            p_htseq.communicate()[0]
        if p_htseq.returncode != 0:
            if sortedbam_fh is not None:
                os.unlink(sortedbam_fh.name)
            raise subprocess.CalledProcessError(p_htseq.returncode, cmd, None)
        if sortedbam_fh is not None:
```

```
            os.unlink(sortedbam_fh.name)
        return (cmd, p_htseq.returncode)
```

5.1.2 Steps performing several activities

Slightly more work than for the simple case will be required, but not too much either.

5.2 Unified execution

A new script, extending or overriding the default execution layer in the package can be written very simply:

```
from railroadtracks import core, rnaseq

class MyStep(core.Stepabstract):
   # Definition of a new step
   pass

NEW_STEPLIST_CLASSES = list()
for cls in rnaseq._STEPLIST_CLASSES:
    NEW_STEPLIST.append(cls)
NEW_STEPLIST_CLASS.append(MyStep)

if __name__ == '__main__':
    # use the command and subcommands parsing
    # with the new list of steps
    core.unified_exec(classlist = NEW_STEPLIST_CLASSES)
```

CHAPTER

SIX

UNIFIED EXECUTION

Almost each tool in a sequence of steps used for RNA-Seq processing is using idiosyncratic parameters or arguments. This is making the task of a person wanting to use them both prone to errors (as these tools are also not always checking thoroughly that the input and parameters make sense) and to a significant time spent reading scattered sources of information (the documentation for the tools if often insufficient and much of the the knowledge about them is spread throughout internet forums and mailing-lists)

```
$ alias unifex='python -m railroadtracks.unifex'
```

Three modes exist: *run*, *version*, and *activities*

```
$ unifex
usage: unifex.py [-h] {run,version,activities} ...
rnaseq.py: error: too few arguments
```

A step is then defined by the name of the executable (either as a name to be found in the *${PATH}*, or as an absolute path), as well as by the name of a modeling class.

Note: Specifying both the executable and the modeling class is required because a number of executables can perform different tasks/activities.

For example, the command STAR can be used to build an index reference for subsequent alignment, or perform an alignment. Specifying the model associated with the executable helps to remove the ambiguity.

```
$ unifex run star-index STAR
$ unifex run star-align STAR # astrology anyone ?
```

6.1 Version number

Obtaining the version number is achieved the same way for all steps

```
$ unifex version bowtie2-build bowtie2
2.1.0
$ unifex version star-align STAR
STAR_2.3.0e_r291
$ unifex version limma-voom R
3.18.3
$ unifex version edger R
3.4.0
```

6.2 Running a step

Running a step is also achieved essentially the same way for all steps. All steps have "sources" (that is input files) and "targets" (that is destination/output files).

```
$ # bowtie2 (create index)
$ unifex run bowtie2-build bowtie2 \
      -s <name>=<source file(s)> \
      -t <name>=<target file(s)>
$ # STAR (create index)
$ unifex run star-index STAR \
      -s <name>=<source file(s)> \
      -t <name>=<target file(s)>
```

Note: Whenever the sources and targets expected by a given tool are not specified, the bash command fails and print the list of missing parameters

```
$ unifex run edger R
The following sources must be defined (and are missing):
- counttable_fn
- sampleinfo_fn
The following targets must be defined (and are missing):
- diffexp_fn
```

6.3 Using a scheduler/Queueing system

The persistent layer can generate the bash commands for running the unified execution. This is making the use of any existing queuing or scheduling system able to take *bash* script straightforward.

Note: This is possible, yet not fully documented. Please check the examples of recipes (Section *Recipes*) for details about how it is already possible to run tasks from a Python script.

6.4 Docstrings

Unified execution layer.

One general way to run things on the command line.

class `railroadtracks.unifex.`**`Call`**(*step*, *assets*, *parameters*)

Unified call, turning a step + assets + parameters into a task.

`execute`()

Execute the task.

`railroadtracks.unifex.`**`build_AssetSet`**(*AssetSet*, *values*)

Parameters

- **AssetSet** –
- **values** – values to create instances in the AssetSet

`railroadtracks.unifex.`**`unified_exec_run`**(*args*, *steplist*, *msg=[]*)

Run a command. :param args: arguments in a class such as the one returned by

`argparse.ArgumentParser.parse()` :param steplist: sequence of known steps. The *args* will be matched against this to find the model class. :type steplist: sequence of `core.StepAbstract`-inherting instances :param msg: list with (eventual) messages

CHAPTER

SEVEN

PERSISTENCE

Note: This part of the documentation should only concern users interested in moving the persistence layer to a different system to store metadata associated with commands performed.

Tasks to be performed are stored persistently on disk. This is required to ensure that all steps computed, and the sequence of steps leading to results, are conserved across restarts of the main process or of the system.

Currently, this is implemented in an SQLite database. Persistence/memoization for the DAG

class `railroadtracks.hortator.`**DbID**
DbID(id, new)

id
Alias for field number 0

new
Alias for field number 1

class `railroadtracks.hortator.`**PersistentTaskList**(*db_fn*, *model*, *wd='.'*, *force_create=False*)
List of tasks stored on disk.

class StoredEntityNoLabel
StoredEntityNoLabel(id, clsname, entityname)

clsname
Alias for field number 1

entityname
Alias for field number 2

id
Alias for field number 0

`PersistentTaskList.`**finalsteps**()
Concrete steps for which all targets are final

`PersistentTaskList.`**get_parenttask_of_storedentity**(*stored_entity*)
Return the task producing a stored entity. There should obviously only be one such task, and an Exception is raised if not the case. :param stored_entity: the stored entity in the database :type stored_entity: `StoredEntity` :rtype: a *StepConcrete_DbEntry* `namedtuple`, or None

`PersistentTaskList.`**get_sourcesofactivity**(*activity*)
Retrieve the sources of steps performing a specific activity. :param activity: an activity :type activity: `Enum`

PersistentTaskList.**get_srcassets**(*concrete_step_id*)
Return the source files for a given concrete step ID. :param concrete_step_id: ID for the concrete step in the database. :rtype: generator

PersistentTaskList.**get_targetassets**(*concrete_step_id*)
Return the target files for a given concrete step ID. :param concrete_step_id: ID for the concrete step in the database. :rtype: generator

PersistentTaskList.**get_targetsofactivity**(*activity*)
Retrieve the targets of steps performing a specific activity. :param activity: an activity :type activity: Enum

PersistentTaskList.**get_targetsoftype**(*clsname*)
Return all targets of a given type.

PersistentTaskList.**get_targetstepconcrete**(*stored_entity*)
Return the tasks using a given stored entity. :param stored_entity: the stored entity in the database. :type stored_entity: can be StoredEntity or StoredSequence :rtype: a *SepConcrete_DbEntry* namedtuple, or None

PersistentTaskList.**id_step_activity**(*activity*)
Conditionally add an activity (add only if not already present) :param activity: one actibity name :rtype: ID for the activity as an integer

PersistentTaskList.**id_step_type**(*activities*)
Conditionally add a step type (add only if not already present). :param activities: sequence of activity names :rtype: ID for the step type as an integer

PersistentTaskList.**id_step_variant**(*step*, *activities*)
Return a database ID for the step variant (creating a new ID only of the variant is not already tracked)

Parameters

- **step** (core.StepAbstract) – a step
- **activities** – a sequence of activity names

Return type ID for a step variant as an int[1].

PersistentTaskList.**id_stepconcrete**(*step_variant_id*, *sources*, *targets*, *parameters*, *tag=1*)
Conditionally add a task ("concrete" step), that is a step variant (executable and parameters) to which source and target files, as well as parameters, are added.

Parameters

- **step_variant_id** (*integer*) – ID for the step variant
- **sources** (AssetSet) – sequence of sources
- **targets** (AssetSet) – sequence of targets
- **parameters** (a sequence of str[2]) – list of parameters
- **tag** (a sequence of int[3]) – a tag, used to performed repetitions of the exact same task

Return type DbID

PersistentTaskList.**id_stepparameters**(*parameters*)
Conditionally add parameters (add only if not already present) :param parameters: sequence of parameters :rtype: ID for the pattern as a DbID.

[1] http://docs.python.org/library/functions.html#int
[2] http://docs.python.org/library/functions.html#str
[3] http://docs.python.org/library/functions.html#int

PersistentTaskList.**id_stored_entity**(*cls*, *name*)
Conditionally add a stored entity (add only if not already present) :param cls: Python class for the stored entity :param name: Parameter "name" for the class "cls". :rtype: ID for the pattern as a DbID.

PersistentTaskList.**id_stored_sequence**(*cls*, *clsname_sequence*)
Conditionally add a stored entity (add only if not already present) :param clsname_sequence: Sequence of pairs (Python class for the stored entity, parameter "name" for the class "cls") :rtype: ID for the pattern as a DbID.

PersistentTaskList.**iter_finaltargets**()
Targets not used as source anywhere else.

PersistentTaskList.**iter_steps**()
Iterate through the concrete steps

PersistentTaskList.**statuslist**
Status list

PersistentTaskList.**version**
Version for the database and package (mixing versions comes at one's own risks)

class railroadtracks.hortator.**Step**(*step*, *sources*, *targets*, *parameters*, *model*)
When used in the context of a StepGraph, a Step is small graph, or subgraph, consituted of a vertex, connected downstream to targets and upstream to sources. For more information about a StepGraph, see the documentation for it.

class railroadtracks.hortator.**StepGraph**(*cache*)
The steps to be performed are stored in a directed acyclic graph (DAG).

This graph can be thought of as a two-level graph. The higher level represents the connectivity between steps (we will call supersteps), and the lower-level expands each step into sources, targets, and a step using sources to produce targets.

There is a persistent representation (currently a mysql database), and this class is aiming at isolating this implementation detail from a user.

add(*step*, *assets*, *parameters=()*, *tag=1*)
Add a step, associated assets, and optional parameters, to the StepGraph.

The task graph is like a directed (presumably) acyclic multilevel graph. Asset vertices are only connected to step vertices (in other words asset vertices represent connective layers between steps).

Parameters

- **step** (a core.StepAbstract (or of child classes) object) – The step to be added
- **assets** (a core.AssetStep (or of child classes) object) – The assets linked to the step added. If assets.target is undefined, the method will define it with unique identifiers and these will assigned in place.
- **parameters** (A sequence of str[4] elements) – Parameters for the step
- **tag** – A tag to differentiate repetitions of the exact same task.

Return type StepConcrete_DbEntry as the entry added to the database

cleantargets_stepconcrete(*step_concrete_id*)
Clean the targets downstream of a task (step_concrete), which means erasing the target files and (re)setting the tasks' status to 'TO DO'.

Parameters **step_concrete_id** – A task

[4] http://docs.python.org/library/functions.html#str

destinationwalk_stepconcrete(*step_concrete_id*, *func_stored_entity*, *func_stored_sequence*, *func_step_concrete*, *func_storedentity_stepconcrete*, *func_stepconcrete_storedentity*)

Walk down the path.

destinationwalk_storedentity(*stored_entity*, *func_stored_entity*, *func_stored_sequence*, *func_step_concrete*, *func_storedentity_stepconcrete*, *func_stepconcrete_storedentity*)

Walk down the path.

provenancewalk_storedentity(*stored_entity*, *func_stored_entity*, *func_stored_sequence*, *func_step_concrete*, *func_storedentity_stepconcrete*, *func_stepconcrete_storedentity*)

Walk up the path. :param stored_entity_id: the stored entity to start from :param func_stored_entity: a callback called with each stored entity :param func_step_concrete: a callback called with each step concrete :param func_storedentity_stepconcrete: a callback called with each link between a stored entity and a step concrete :param func_stepconcrete_storedentity: a callback called with each link between a step concrete and a stored entity

static **stepconcrete_dirname**(*stepconcrete_id*)

Name of the directory corresponding to an ID.

Parameters **stepconcrete_id** – ID for a directory.

railroadtracks.hortator.**TaskStatusCount**

alias of TaskStatus

CHAPTER

EIGHT

INDEX AND TABLES

- *genindex*
- *modindex*
- *search*

r

Symbols

A

B

C

D

E

F

G

I

M

N

P

R

S

T

U

V

W