dammit Documentation

Release 1.0rc2

Camille Scott

Jul 09, 2018

Contents

dammit is a simple de novo transcriptome annotator. It was born out of the observations that annotation is mundane and annoying, all the individual pieces of the process exist already, and the existing solutions are overly complicated or rely on crappy non-free software.

Science shouldn't suck for the sake of sucking, so dammit attempts to make this sucky part of the process suck a little less.

dammit is free and open source, and has been built around a free and open source ecosystem. As such, programs which the author does not consider free enough have been eschewed as dependencies. This can either mean programs with non-free licenses *or* programs which are overly difficult to install and configure – we believe that access is a part of openness.

Fig. 1: *Your PI, wistfully thinking back on Perl 4*

CHAPTER 1

Details

Authors Camille Scott

Contact camille.scott.w@gmail.com

GitHub https://github.com/camillescott/dammit

License BSD

Citation `bibtex`

CHAPTER 2

Topics

2.1 README

dammit is a simple de novo transcriptome annotator. It was born out of the observation that: annotation is mundane and annoying; all the individual pieces of the process exist already; and, the existing solutions are overly complicated or rely on crappy non-free software.

Science shouldn't suck for the sake of sucking, so dammit attempts to make this sucky part of the process suck a little less.

2.1.1 System Requirements

dammit, for now, is officially supported on GNU/Linux systems via bioconda. macOS support will be available via bioconda soon.

For the standard pipeline, dammit needs ~18GB of space to store its prepared databases, plus a few hundred MB per BUSCO database. For the standard annotation pipeline, I recommended 16GB of RAM. This can be reduced by editing LAST parameters via a custom configuration file.

The full pipeline, which uses uniref90, needs several hundred GB of space and considerable RAM to prepare the databases.

2.1.2 Installation

As of version 1.*, the recommended installation platform for dammit is via bioconda. If you already have anaconda installed, proceed to the next step. Otherwise, you can either follow the instructions from bioconda, or if you're on Ubuntu (or most GNU/Linux platforms), install it directly into your home folder with:

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O␣
↪miniconda.sh && bash miniconda.sh -b -p $HOME/miniconda
echo 'export PATH="$HOME/miniconda/bin:$PATH"' >> $HOME/.bashrc
```

It's recommended that you use *conda* environments to separate your packages, though it isn't strictly necessary:

```
conda create -n dammit python=3
source activate dammit
```

Now, add the channels and install dammit:

```
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda

conda install dammit
```

And that's it!

2.1.3 Usage

To check for databases, run:

```
dammit databases
```

and to download and install the general databases, use:

```
dammit databases --install
```

A reduced database set that excludes OrthoDB, uniref, Pfam, and Rfam (ie, all the homology searches other than user-supplied databases) with:

```
dammit databases --install --quick
```

dammit supports all the released BUSCO databases, which can be installed with the *–busco-group* flag; a complete list of available groups can be seen with *dammit databases -h*:

```
dammit databases --install --busco-group fungi
```

To annotate your transcriptome, the most basic usage is:

```
dammit annotate <transcriptome_fasta>
```

These are extremely basic examples; for a much more detailed description, take a look at the relevant page in the documentation. The documentation describes how to customization the database installation location and utilize existing databases.

2.1.4 Known Issues

- On some systems, installation of the ConfigParser package can get borked, which will cause and exception to be thrown. This can be fixed by following the directions at issue #33: https://github.com/camillescott/dammit/issues/33.
- There can be errors resuming runs which were interrupted on the BUSCO stage. If the task fails on resume, delete the BUSCO results folder within your dammit results folder, which will have a name of the form *run_<name>.busco_results*.

2.1.5 Acknowledgements

I've received input and advice from a many sources, including but probably not limited to: C Titus Brown, Matt MacManes, Chris Hamm, Michael Crusoe, Russell Neches, Luiz Irber, Lisa Cohen, Sherine Awad, and Tamer Mansour.

CS was funded by the National Human Genome Research Institute of the National Institutes of Health under Award Number R01HG007513 through May 2016, and now receives support from the Gordon and Betty Moore Foundation under Award number GBMF4551.

2.2 Installation

2.2.1 Non-python Dependencies

First we will take care of the external non-python dependencies; then we'll move on to getting our python environment ready.

Unfortunately, annotation necessarily relies on many software packages. I have worked hard to make dammit rely only on software which is accessible *and* likely to continue to be so. Most of the dependencies are available in either Ubuntu PPAs or PyPI, and if not, are trivial to install manually. If the goal is to make annotation suck less, then installing the necessary software should suck less too.

Most of this guide will assume you're on a Ubuntu system. However, the dependencies should all run on any flavor of GNU/Linux and on OSX.

First, let's get packages from the Ubuntu PPAs:

```
sudo apt-get update
sudo apt-get install git ruby hmmer unzip build-essential \
    infernal ncbi-blast+ liburi-escape-xs-perl emboss liburi-perl \
    libsm6 libxrender1 libfontconfig1 parallel
```

If you're on Ubuntu 15.10, you can also install TransDecoder and LAST through aptitude:

```
sudo apt-get install transdecoder last-align
```

Otherwise, you'll need to install them manually. To install TransDecoder in your home directory, execute these commands in your terminal:

```
cd
curl -LO https://github.com/TransDecoder/TransDecoder/archive/2.0.1.tar.gz
tar -xvzf 2.0.1.tar.gz
cd TransDecoder-2.0.1; make
export PATH=$HOME/TransDecoder-2.0.1:$PATH
```

To get LAST:

```
cd
curl -LO http://last.cbrc.jp/last-658.zip
unzip last-658.zip
cd last-658
make
export PATH=$HOME/last-658/src:$PATH
export PATH=$HOME/last-658/scripts:$PATH
```

The above commands will only install them for the current session; to keep it installed, append the exports to your bash profile:

```
echo 'export PATH=$HOME/TransDecoder-2.0.1:$PATH' >> $HOME/.bashrc
echo 'export PATH=$HOME/last-658/src:$PATH' >> $HOME/.bashrc
echo 'export PATH=$HOME/last-658/scripts:$PATH' >> $HOME/.bashrc
```

Next, we need to install Conditional Reciprocal Best-hits Blast (CRBB). The algorithm is described in Aubry et al., and is implemented in ruby. Assuming you have ruby (which was installed above), it can be installed with:

```
sudo gem install crb-blast
```

dammit also runs BUSCO to assess completeness. To install it, run the following commands:

```
cd
curl -LO http://busco.ezlab.org/v1/files/BUSCO_v1.22.tar.gz
tar -xvzf BUSCO_v1.22.tar.gz
chmod +x BUSCO_v1.22/*.py
export PATH=$HOME/BUSCO_v1.22:$PATH
```

. . . and once again, to install it permanently:

```
echo 'export PATH=$HOME/BUSCO_v1.22:$PATH' >> $HOME/.bashrc
```

2.2.2 Python Dependencies

dammit is a python package, and relies on a number of commonly-used scientific libraries. If you're sure you have the following python dependencies already, you can skip this step and move on to the final stage:

```
setuptools>=0.6.35
pandas>=0.17
khmer>=2.0
doit>=0.29.0
nose==1.3.4
ficus>=0.1
matplotlib>=1.0
```

Otherwise, we will have to install them. Pandas, numpy, and matplotlib are quite hefty, mostly because they require a lot of compilation. To get around this, you can either install them via Anaconda, which I recommend, or you can install those which are available through the Ubuntu PPAs. If you wish to do things the slow but traditional way, you can just skip right ahead and:

```
pip install -U setuptools
pip install dammit
```

Otherwise, proceed to the Anaconda instructions, or skip ahead to the hybrid *Ubuntu / Pip Instructions*.

Anaconda

Anaconda (or miniconda) is the preferred distribution for dammit. It's straightforward to install and saves a lot of time compiling things when creating new environments. To install it on Ubuntu, first download it:

```
cd
curl -OL https://3230d63b5fc54e62148e-c95ac804525aac4b6dba79b00b39d1d3.ssl.cf1.
↪rackcdn.com/Anaconda2-4.0.0-Linux-x86_64.sh
```

And run the installer:

```
bash Anaconda2-2.4.0-Linux-x86_64.sh -b
echo 'export PATH=$HOME/anaconda2/bin:$PATH' >> $HOME/.bashrc
```

Select *yes* when prompted on adding it to your *.bashrc*, and resource your profile to gain access to it:

```
source .bashrc
```

The version of Sphinx which is shipped with Anaconda has issues; we will remove it and allow dammit to install its own version via PyPI:

```
conda remove sphinx
```

Get the latest versions of some packages:

```
conda update pandas numexpr
```

Ubuntu / Pip Instructions

If you'd prefer to not use Anaconda, are on a clean Ubuntu 14.04 machine, have not installed the python packages with pip, and have installed the non-python dependencies, you can install them through the Ubuntu PPAs as follows:

```
sudo apt-get update
sudo apt-get install python-pip python-dev python-numpy
```

Unfortunately, you'll still have to install Pandas through pip, as the versions in the Ubuntu 14.04 PPAs are quite old. These will be installed automatically along with dammit.

Dammit

dammit itself is quite easy to install. Just run:

```
pip install -U setuptools
pip install dammit
```

If you're not running anaconda or a virtual environment, you'll have to put a *sudo* before pip to install it globally. If you don't already have a recent versions of Pandas and scikit-learn this will take a bit.

When you're done, run the check again to make sure everything was installed correctly:

```
dammit dependencies
```

And you're ready to go!

2.3 Tutorial

Once you have the dependencies installed, it's time to actually annotate something! This guide will take you through a short example on some test data.

2.3.1 Data

First let's download some test data. We'll start small and use a *Schizosaccharomyces pombe* transcriptome. Make a working directory and move there, and then download the file:

```
mkdir dammit_test
cd dammit_test
wget ftp://ftp.ebi.ac.uk/pub/databases/pombase/FASTA/cdna_nointrons_utrs.fa.gz
wget ftp://ftp.ebi.ac.uk/pub/databases/pombase/FASTA/pep.fa.gz
```

Decompress the file with gunzip:

```
gunzip cdna_nointrons_utrs.fa.gz pep.fa.gz
```

2.3.2 Databases

If you're just starting, you probably haven't downloaded the databases yet. Here we'll install the main databases, as well as the *eukaryota* BUSCO database for our yeast dataset. This could take a while, so consider walking away and getting yourself a cup of coffee. If you installed dammit into a virtual environment, be sure to activate it first:

```
dammit databases --install --busco-group eukaryota
```

Alternatively, if you happen to have downloaded many of these databases before, you can follow the directions in the databases guide.

While the initial download takes a while, once its done, you won't need to do it again – dammit keeps track of the database state and won't repeat work its already completed, even if you accidentally rerun with the `--install` flag.

2.3.3 Annotation

Now we'll do a simple run of the annotator. We'll use *pep.fa* as a user database; this is a toy example, seeing as these proteins came from the same set of transcripts as we're annotating, but they illustrate the usage nicely enough. We'll also specify a non-default BUSCO group. You can replace the argument to `--n_threads` with however many cores are available on your system in order to speed it up.:

```
dammit annotate cdna_nointrons_utrs.fa --user-databases pep.fa --busco-group␣
↪eukaryota --n_threads 1
```

This will take a bit, so go get another cup of coffee. . .

2.4 Usage

If you're looking for a quick start, head over to the tutorial. This page has more complete usage information and a better breakdown of the functionality.

2.4.1 Dependencies

dammit has three components. The first, *dependencies*, checks whether you have the dependencies installed correctly and warns you if not. It is run with:

```
dammit dependencies
```

There isn't much to this command; either you have the dependencies or you don't. If you don't, there are instructions for getting them on the installation page.

2.4.2 Databases

The next component is the *databases* subcommand. This handles all of dammit's external data; the documentation can be found here.

2.4.3 Annotation

The *annotate* command runs the BUSCO assessment, assembly stats, and homology searches, aggregates the results, and outputs a GFF3 file and annotation report. It takes the `--full`, `--database-dir`, and `--busco-group` options in the same manner as the *databases* command. Additionally, it can specify an optional output directory, the number of threads to use with threaded subprograms like HMMER, and a list of user-supplied protein databases in FASTA format. A simple invocation with the default databases would look like:

```
dammit annotate <transcriptome.fasta>
```

While a more complex invocation might look like:

```
dammit annotate <transcriptome.fasta> --database-dir /path/to/dbs --busco-group␣
↪vertebrata --n_threads 4 --user-databases whale.pep.fasta dolphin.pep.fasta
```

User databases will be searched with CRBB; this runs *blastx*, so if you supply ridiculously huge databases, it *will* take a long time. Future versions will use LAST for all searches to improve performance, but for now, we're stuck with the NCBI's dinosaur. Also note that the information from the deflines in your databases will be used to construct the GFF3 file, so if your databases lack useful IDs, your annotations will too.

2.5 Databases

2.5.1 Basic Usage

dammit handles databases under the `dammit databases` subcommand. By default, dammit looks for databases in *$HOME/.dammit/databases* and will install them there if missing. If you have some of the databases already, you can inform dammit with the `--database-dir` flag.

To check for databases in the default location:

```
dammit databases
```

To check for them in a custom location, you can either use the *–database-dir* flag:

```
dammit databases --database-dir /path/to/databases
```

or, you can set the *DAMMIT_DB_DIR* environment variable. The flag will supersede this variable, falling back to the default if neither is set. For example:

```
export DAMMIT_DB_DIR=/path/to/databases
```

This can also be added to your *$HOME/.bashrc* file to make it persistent.

To download and install them into the default directory:

```
dammit databases --install
```

For more details, check out the Advanced-Database-Handling section.

2.5.2 About

dammit uses the following databases:

1. Pfam-A

   Pfam-A is a collection of protein domain profiles for use with profile hidden markov model programs like hmmer. These searches are moderately fast and very sensitive, and the Pfam database is very well curated. Pfam is used during TransDecoder's ORF finding and for annotation assignment.

2. Rfam

   Rfam is a collection of RNA covariance models for use with programs like Infernal. Covariance models describe RNA secondary structure, and Rfam is a curated database of non-coding RNAs.

3. OrthoDB

   OrthoDB is a curated database of orthologous genes. It attempts to classify proteins from all major groups of eukaryotes and trace them back to their ancestral ortholog.

4. BUSCO

   BUSCO databases are collections of "core" genes for major domains of life. They are used with an accompanying BUSCO program which assesses the completeness of a genome, transcriptome, or list of genes. There are multiple BUSCO databases, and which one you use depends on your particular organism. Currently available databases are:

   (a) Metazoa

   (b) Vertebrata

   (c) Arthropoda

   (d) Eukaryota

   dammit uses the metazoa database by default, but different databases can be used with the `--busco-group` parameter. You should try to use the database which most closely bounds your organism.

5. uniref90

   uniref is a curated collection of most known proteins, clustered at a 90% similarity threshold. This database is comprehensive, and thus quite enormous. dammit does not include it by default due to its size, but it can be installed and used with the `--full` flag.

A command using all of these potential options and databases might look like:

```
dammit databases --install --database-dir /path/to/dbs --full --busco-group arthropoda
```

2.5.3 Advanced Database Handling

Several of these databases are quite large. Understandably, you probably don't want to download or prepare them again if you already have. There are a few scenarios you might run in to.

1. You already have the databases, and they're all in one place and properly named.

   Excellent! This is the easiest. You can make use of dammit's `--database-dir` flag to tell it where to look. When running with `--install`, it will find the existing files and prep them if necessary.:

   ```
   dammit databases --database-dir <my_database_dir> --install
   ```

2. Same as above, but they have different names.

dammit expects the databases to be "properly" named – that is, named the same as their original forms. If your databases aren't named the same, you'll need to fix them. But that's okay! We can just soft link them. Let's say you have Pfam-A already, but for some reason its named *all-the-models.hmm*. You can link them to the proper name like so:

```
cd <my_database_dir>
ln -s all-the-models.hmm Pfam-A.hmm
```

If you already formatted it with *hmmpress*, you can avoid repeating that step as well:

```
ln -s all-the-models.hmm.h3f Pfam-A.hmm.h3f
ln -s all-the-models.hmm.h3i Pfam-A.hmm.h3i
ln -s all-the-models.hmm.h3m Pfam-A.hmm.h3m
ln -s all-the-models.hmm.h3p Pfam-A.hmm.h3p
```

For a complete listing of the expected names, just run the `databases` command:

```
dammit databases
```

3. You have the databases, but they're scattered to the virtual winds.

   The fix here is similar to the above. This time, however, we'll soft link all the databases to one location. If you've run `dammit databases`, a new directory will have been created at *$HOME/.dammit/databases*. This is where they are stored by default, so we might as well use it! For example:

   ```
   cd $HOME/.dammit/databases
   ln -s /path/to/all-the-models.hmm Pfam-A.hmm
   ```

   And repeat for all the databases. Now, in the future, you will be able to run dammit without the *–database-dir* flag.

Alternatively, if this all seems like too much of a hassle and you have lots of hard drive space, you can just say "to hell with it!" and reinstall everything with:

```
dammit databases --install
```

2.6 API Docs

2.6.1 Subpackages

dammit.fileio package

Submodules

dammit.fileio.base module

class dammit.fileio.base.**BaseParser**(*filename*)
Bases: object

raise_empty()

class dammit.fileio.base.**ChunkParser**(*filename*, *chunksize=10000*)
Bases: *dammit.fileio.base.BaseParser*

empty()
Get an empty DataFrame with the appropriate columns.

read()
Read the entire file at once and return as a single DataFrame.

exception dammit.fileio.base.**EmptyFile**
Bases: Exception

dammit.fileio.base.**convert_dtypes**(*df*, *dtypes*)
Convert the columns of a DataFrame to the types specified in the given dictionary, inplace.

Parameters

- **df** (*DataFrame*) – The DataFrame to convert.
- **dtypes** (*dict*) – Dictionary mapping columns to types.

dammit.fileio.base.**next_or_raise**(*fp*)
Get the next line and raise an exception if its empty.

dammit.fileio.base.**warn_empty**(*msg*)
Warn that a file is empty.

dammit.fileio.gff3 module

class dammit.fileio.gff3.**GFF3Parser**(*filename*, ***kwargs*)
Bases: *dammit.fileio.base.ChunkParser*

columns = [('seqid', <class 'str'>), ('source', <class 'str'>), ('type', <class 'str'>

static decompose_attr_column(*col*)

empty()
Get an empty DataFrame with the appropriate columns.

class dammit.fileio.gff3.**GFF3Writer**(*filename=None*, *converter=None*, ***converter_kwds*)
Bases: object

convert(*data_df*)

static mangle_coordinates(*gff3_df*)
Although 1-based fully closed intervals are of the Beast, we will respect the convention in the interests of peace between worlds and compatibility.

Parameters **gff3_df** (*DataFrame*) – The DataFrame to "fix".

version_line = '##gff-version 3.2.1'

write(*data_df*, *version_line=True*)
Write the given data to a GFF3 file, using the converter if given.

Generates an empty file if given an empty DataFrame.

Parameters

- **version_line** (*bool*) – If True, write the GFF3 version line at the.
- **that this will cause an existing file to be overwritten, but** (*Note*) –
- **only be added in the first call to write.** (*will*) –

dammit.fileio.gff3.**cmscan_to_gff3**(*cmscan_df*, *tag=''*, *database=''*)

dammit.fileio.gff3.**hmmscan_to_gff3**(*hmmscan_df*, *tag=''*, *database=''*)

dammit.fileio.gff3.**id_gen_wrapper**()

dammit.fileio.gff3.**maf_to_gff3**(*maf_df*, *tag=''*, *database=''*, *ftype='translated_nucleotide_match'*)

Convert a MAF DataFrame to a GFF3 DataFrame ready to be written to disk.

Parameters

- **maf_df** (*pandas.DataFrame*) – The MAF DataFrame. See dammit.fileio.maf.MafParser for column specs.
- **tag** (*str*) – Extra tag to add to the source column.
- **database** (*str*) – For the database entry in the attributes column.
- **ftype** (*str*) – The feature type; GMOD compliant if possible.

Returns The GFF3 compliant DataFrame.

Return type pandas.DataFrame

dammit.fileio.gff3.**next_ID**()

dammit.fileio.gff3.**shmlast_to_gff3**(*df*, *database=''*)

dammit.fileio.hmmer module

class dammit.fileio.hmmer.**HMMerParser**(*filename*, *query_regex=None*, *query_basename='Transcript'*, ***kwargs*)

Bases: *dammit.fileio.base.ChunkParser*

columns = [('target_name', <class 'str'>), ('target_accession', <class 'str'>), ('tlen

dammit.fileio.infernal module

class dammit.fileio.infernal.**InfernalParser**(*filename*, ***kwargs*)

Bases: *dammit.fileio.base.ChunkParser*

columns = [('target_name', <class 'str'>), ('target_accession', <class 'str'>), ('query

dammit.fileio.maf module

class dammit.fileio.maf.**MafParser**(*filename*, *aln_strings=False*, *chunksize=10000*, ***kwargs*)

Bases: *dammit.fileio.base.ChunkParser*

columns = [('E', <class 'float'>), ('EG2', <class 'float'>), ('q_aln_len', <class 'int

Module contents

dammit.tasks package

Submodules

dammit.tasks.busco module

class dammit.tasks.busco.**BuscoTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*input_filename*, *output_name*, *busco_db_dir*, *input_type='tran'*, *n_threads=1*, *config_file=None*, *params=None*)
Get a task to run BUSCO on the given FASTA file.

Parameters

- **input_filename** (*str*) – The FASTA file to run BUSCO on.
- **output_name** (*str*) – Base name for the BUSCO output directory.
- **busco_db_dir** (*str*) – Directory with the BUSCO databases.
- **input_type** (*str*) – By default, *trans* for transcriptome.
- **n_threads** (*int*) – Number of threads to use.
- **params** (*list*) – Extra parameters to pass to the executable.

Returns A doit task.

Return type dict

dammit.tasks.busco.**busco_to_df**(*fn_list, dbs=['metazoa', 'vertebrata']*)
Given a list of BUSCO results from different databases, produce an appropriately multi-indexed DataFrame of the results.

Parameters

- **fn_list** (*list*) – The BUSCO summary files.
- **dbs** (*list*) – The BUSCO databases used for these runs.

Returns The BUSCO results.

Return type DataFrame

dammit.tasks.busco.**parse_busco_full**(*fn*)
Parses a BUSCO full result table into a Pandas DataFrame.

Parameters **fn** (*str*) – The results file.

Returns The results DataFrame.

Return type DataFrame

dammit.tasks.busco.**parse_busco_multiple**(*fn_list, dbs=['metazoa', 'vertebrata']*)
Parses multiple BUSCO results summaries into an appropriately index DataFrame.

Parameters

- **fn_list** (*list*) – List of paths to results files.
- **dbs** (*list*) – List of BUSCO database names.

Returns The formated DataFrame.

Return type DataFrame

dammit.tasks.busco.**parse_busco_summary**(*fn*)
Parses a BUSCO summary file into a JSON compatible dictionary.

Parameters **fn** (*str*) – The summary results file.

Returns The BUSCO results.

Return type dict

dammit.tasks.fastx module

dammit.tasks.fastx.**get_rename_transcriptome_task**(*transcriptome_fn*, *output_fn*, *names_fn*, *transcript_basename*, *split_regex=None*)
Create a doit task to copy a FASTA file and rename the headers.

Parameters

- **transcriptome_fn** (*str*) – The FASTA file.
- **output_fn** (*str*) – Destination to copy to.
- **names_fn** (*str*) – Destination to the store mapping from old to new names.
- **transcript_basename** (*str*) – String to contruct new names from.
- **split_regex** (*regex*) – Regex to split the input names with; must contain a *name* field.

Returns A doit task.

Return type dict

dammit.tasks.fastx.**get_transcriptome_stats_task**(*transcriptome*, *output_fn*)
Create a doit task to run basic metrics on a transcriptome.

Parameters

- **transcriptome** (*str*) – The input FASTA file.
- **output_fn** (*str*) – File to store the results.

Returns A doit task.

Return type dict

dammit.tasks.fastx.**strip_seq_extension**(*fn*)

dammit.tasks.gff module

dammit.tasks.gff.**get_cmscan_gff3_task**(*input_filename*, *output_filename*, *database*)
Given raw input from Infernal's cmscan, convert it to GFF3 and save the results.

Parameters

- **input_filename** (*str*) – The input CSV.
- **output_filename** (*str*) – Destination for GFF3 output.
- **database** (*str*) – Tag to use in the GFF3 *Dbxref* field.

Returns A doit task.

Return type dict

dammit.tasks.gff.**get_gff3_merge_task**(*gff3_filenames*, *output_filename*)
Given a list of GFF3 files, merge them all together.

Parameters

- **gff3_filenames** (*list*) – Paths to the GFF3 files.
- **output_filename** (*str*) – Path to pipe the results.

Returns A doit task.

Return type dict

dammit.tasks.gff.**get_hmmscan_gff3_task**(*input_filename*, *output_filename*, *database*)
Given HMMER output converted to CSV, convert it to GFF3 and save the results. CSV generated from the DataFrame(s) returned by the HMMerParser.

Parameters

- **input_filename** (*str*) – The input CSV.
- **output_filename** (*str*) – Destination for GFF3 output.
- **database** (*str*) – Tag to use in the GFF3 *Dbxref* field.

Returns A doit task.

Return type dict

dammit.tasks.gff.**get_maf_best_hits_task**(*maf_fn*, *output_fn*)
Doit task to get the best hits from a lastal MAF file.

Parameters

- **maf_fn** (*str*) – Path to the MAF file.
- **output_fn** (*str*) – Path to store resulting CSV file.

Returns A doit task.

Return type dict

dammit.tasks.gff.**get_maf_gff3_task**(*input_filename*, *output_filename*, *database*)
Given either a raw MAF file or a CSV file with the proper MAF colums, convert it to GFF3 and save the results.

Parameters

- **input_filename** (*str*) – The input MAF or CSV.
- **output_filename** (*str*) – Destination for GFF3 output.
- **database** (*str*) – Tag to use in the GFF3 *Dbxref* field.

Returns A doit task.

Return type dict

dammit.tasks.gff.**get_shmlast_gff3_task**(*input_filename*, *output_filename*, *database*)
Given the CSV output from shmlast, convert it to GFF3 and save the results.

Parameters

- **input_filename** (*str*) – The input CSV.
- **output_filename** (*str*) – Destination for GFF3 output.
- **database** (*str*) – Tag to use in the GFF3 *Dbxref* field.

Returns A doit task.

Return type dict

dammit.tasks.hmmer module

class dammit.tasks.hmmer.**HMMPressTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*db_filename*, *params=None*, *task_dep=None*)
Run hmmpress on a profile HMM database.

Parameters

- **db_filename** (*str*) – The database to run on.
- **params** (*list*) – Extra parameters to pass to executable.
- **task_dep** (*str*) – Task dep to add to doit task.

Returns A doit task.

Return type dict

class dammit.tasks.hmmer.**HMMScanTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*input_filename*, *output_filename*, *db_filename*, *cutoff=1e-05*, *n_threads=1*, *sshloginfile=None*, *params=None*)
Run HMMER's hmmscan with the given database on the given FASTA file.

Parameters

- **input_filename** (*str*) – The path to the input FASTA.
- **output_filename** (*str*) – Path to save the results.
- **db_filename** (*str*) – Path to the formatted database.
- **cutoff** (*float*) – The e-value cutoff to filter with.
- **n_threads** (*int*) – Number of threads to use.
- **pbs** (*bool*) – If True, pass the right parameters to gnu-parallel to run on a cluster.
- **params** (*list*) – Extra parameters to pass to executable.

Returns A doit task.

Return type dict

dammit.tasks.hmmer.**get_remap_hmmer_task**(*hmmer_filename*, *remap_gff_filename*, *output_filename*, *transcript_basename='Transcript'*)
Given an hmmscan result from the ORFs generated by *TransDecoder.LongOrfs* and TransDecoder's GFF3, remap the HMMER results so that they refer to the original nucleotide coordinates rather than the translated ORF coordinates. Produces a CSV file with columns matching those in HMMerParser.

Parameters

- **hmmer_filename** (*str*) – Path to the *hmmscan* results.
- **remap_gff_filename** (*str*) – The GFF3 produced by *TransDecoder.LongOrfs*.

- **output_filename** (*str*) – Path to store remapped results.

Returns A doit task.

Return type dict

dammit.tasks.infernal module

class dammit.tasks.infernal.**CMPressTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*db_filename*, *params=None*, *task_dep=None*)
Run Infernal's *cmpress* on a covariance model database.

Parameters

- **db_filename** (*str*) – Path to the covariance model database.
- **params** (*list*) – Extra parameters to pass to the executable.
- **task_dep** (*str*) – Task dep to give doit task.

Returns A doit task.

Return type dict

class dammit.tasks.infernal.**CMScanTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*input_filename*, *output_filename*, *db_filename*, *cutoff=1e-05*, *n_threads=1*, *sshloginfile=None*, *params=None*)
Run Infernal's *cmscan* on the given FASTA and covariance model database.

Parameters

- **input_filename** (*str*) – Path to the input FASTA.
- **output_filename** (*str*) – Path to store results.
- **db_filename** (*str*) – Path to formatted covariance model database.
- **cutoff** (*float*) – e-value cutoff to filter by.
- **n_threads** (*int*) – Number of threads to run with via gnu-parallel.
- **pbs** (*bool*) – If True, pass parameters to gnu-parallel for running on a cluster.
- **params** (*list*) – Extra parameters to pass to executable.

Returns A doit task.

Return type dict

dammit.tasks.report module

dammit.tasks.report.**generate_sequence_name**(*original_name*, *sequence*, *annotation_df*)

dammit.tasks.report.**generate_sequence_summary**(*original_name*, *sequence*, *annotation_df*)
Given a FASTA sequence's original name, the sequence itself, and a DataFrame with its corresponding GFF3 annotations, generate a summary line of the annotations in key=value format.

Parameters

- **original_name** (*str*) – Original name of the sequence.
- **sequence** (*str*) – The sequence itself.
- **annotation_df** (*DataFrame*) – DataFrame with GFF3 format annotations.

Returns The new summary header.

Return type str

dammit.tasks.report.**get_annotate_fasta_task**(*transcriptome_fn*, *gff3_fn*, *output_fn*)
Annotation the headers in a FASTA file with its corresponding GFF3 file.

Parameters

- **transcriptome_fn** (*str*) – Path to the FASTA file.
- **gff3_fn** (*str*) – Path to the GFF3 annotations.
- **output_fn** (*str*) – Path to store the resulting annotated FASTA.

Returns A doit task.

Return type dict

dammit.tasks.shell module

dammit.tasks.shell.**check_hash**(*target_fn*, *expected*)

dammit.tasks.shell.**get_cat_task**(*file_list*, *target_fn*)
Create a doit task to *cat* together the given files and pipe the result to the given target.

Parameters

- **file_list** (*list*) – The files to *cat*.
- **target_fn** (*str*) – The target file.

Returns A doit task.

Return type dict

dammit.tasks.shell.**get_download_and_gunzip_task**(*url*, *target_fn*)
Create a doit task which downloads and gunzips a file.

Parameters

- **url** (*str*) – URL to download.
- **target_fn** (*str*) – Target file for the download.

Returns doit task.

Return type dict

dammit.tasks.shell.**get_download_and_untar_task**(*url*, *target_dir*, *label=None*)
Create a doit task to download a file and untar it in the given directory.

Parameters

- **url** (*str*) – URL to download.
- **(str** (*target_dir*) – Directory to put the untarred folder in.

- **label** (*str*) – Optional label to resolve doit name conflicts when putting multiple results in the same folder.

Returns doit task.

Return type dict

dammit.tasks.shell.**get_download_task**(*url*, *target_fn*, *md5=None*, *metalink=None*)
Creates a doit task to download the given URL.

Parameters

- **url** (*str*) – URL to download.
- **target_fn** (*str*) – Target for the download.

Returns doit task.

Return type dict

dammit.tasks.shell.**get_gunzip_task**(*archive_fn*, *target_fn*)
Create a doit task to gunzip a gzip archive.

Parameters

- **archive_fn** (*str*) – The gzip file.
- **target_fn** (*str*) – Output filename.

Returns doit task.

Return type dict

dammit.tasks.shell.**get_link_file_task**(*src*, *dst=''*)
Soft-link file to the current directory, or to the destination target if given.

Parameters

- **src** (*str*) – The file to link.
- **dst** (*str*) – The destination; by default, the current directory.

Returns A doit task.

Return type dict

dammit.tasks.shell.**get_untargz_task**(*archive_fn*, *target_dir*, *label=None*)
Create a doit task to untar and gunip a *.tar.gz archive.

Parameters

- **archive_fn** (*str*) – The .tar.gz file.
- **target_dir** (*str*) – The folder to untar into.
- **label** (*str*) – Optional label to resolve doit task name conflicts.

Returns doit task.

Return type dict

dammit.tasks.shell.**hashfile**(*path*, *hasher=None*, *blocksize=65536*)
A function to hash files.

See: http://stackoverflow.com/questions/3431825

dammit.tasks.transdecoder module

class dammit.tasks.transdecoder.**TransDecoderLongOrfsTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*input_filename*, *params=None*)
Get a task to run *Transdecoder.LongOrfs*.

Parameters

- **input_filename** (*str*) – FASTA file to analyze.
- **params** (*list*) – Extra parameters to pass to the executable.

Returns A doit task.

Return type dict

class dammit.tasks.transdecoder.**TransDecoderPredictTask**(*logger=None*)
Bases: *dammit.tasks.utils.DependentTask*

deps()

task(*input_filename*, *pfam_filename=None*, *params=None*)
Get a task to run *TransDecoder.Predict*.

Parameters

- **input_filename** (*str*) – The FASTA file to analyze.
- **pfam_filename** (*str*) – If HMMER has been run against Pfam, pass this file name to *–retain_pfam_hits*.
- **params** (*list*) – Extra parameters to pass to the executable.

Returns A doit task.

Return type dict

dammit.tasks.utils module

class dammit.tasks.utils.**DependentTask**(*logger=None*)
Bases: object

deps()

task(**args*, ***kwargs*)

exception dammit.tasks.utils.**InstallationError**
Bases: RuntimeError

dammit.tasks.utils.**clean_folder**(*target*)
Function for doit task's *clean* parameter to remove a folder.

Parameters **target** (*str*) – The folder to remove.

dammit.tasks.utils.**get_group_task**(*group_name*, *tasks*)
Creat a task group from the given tasks.

Parameters

- **group_name** (*str*) – The name to give the group.

- **tasks** (*list*) – List of Task objects to add to group.

Returns A doit task for the group.

Return type dict

Module contents

2.6.2 Submodules

2.6.3 dammit.annotate module

dammit.annotate.**build_default_pipeline**(*handler*, *config*, *databases*)
Register tasks for the default dammit pipeline.

This is all the main tasks, without lastal uniref90 task.

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

Returns The handler passed in.

Return type *handler.TaskHandler*

dammit.annotate.**build_full_pipeline**(*handler*, *config*, *databases*)
Register tasks for the full dammit pipeline (with uniref90).

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

Returns The handler passed in.

Return type *handler.TaskHandler*

dammit.annotate.**build_nr_pipeline**(*handler*, *config*, *databases*)
Register tasks for the full+nr dammit pipeline (with uniref90 AND nr).

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

Returns The handler passed in.

Return type *handler.TaskHandler*

dammit.annotate.**build_quick_pipeline**(*handler*, *config*, *databases*)
Register tasks for the quick annotation pipeline.

Leaves out the Pfam search (and so does not pass these hits to *TransDecoder.Predict*), the Rfam search, and the lastal searches against OrthoDB and uniref90. Best suited for users who have built their own protein databases and would just like to annotate off them.

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

Returns The handler passed in.

Return type *handler.TaskHandler*

dammit.annotate.**get_handler**(*config*, *databases*)
Build the TaskHandler for the annotation pipelines. The handler will not have registered tasks when returned.

Parameters

- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

Returns A constructed TaskHandler.

Return type *handler.TaskHandler*

dammit.annotate.**register_annotate_tasks**(*handler*, *config*, *databases*)
Register tasks for aggregating the annotations into one GFF3 file and writing out summary descriptions in a new FASTA file.

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.

dammit.annotate.**register_busco_task**(*handler*, *config*, *databases*)
Register tasks for BUSCO. Note that this expects a proper dammit config dictionary.

dammit.annotate.**register_lastal_tasks**(*handler*, *config*, *databases*, *include_uniref=False*, *include_nr=False*)
Register tasks for *lastal* searches. By default, this will just align the transcriptome against OrthoDB; if requested, it will align against uniref90 as well, which takes considerably longer.

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The dictionary of files from a database TaskHandler.
- **include_uniref** (*bool*) – If True, add tasks for searching uniref90.

dammit.annotate.**register_rfam_tasks**(*handler*, *config*, *databases*)
Registers tasks for Infernal's *cmscan* against Rfam. Rfam is an RNA secondary structure database comprising covariance models for many known RNAs. This is a relatively slow step. A proper dammit config dictionary is required.

dammit.annotate.**register_stats_task**(*handler*)
Register the tasks for basic transcriptome metrics.

dammit.annotate.**register_transdecoder_tasks**(*handler*, *config*, *databases*, *include_hmmer=True*)
Register tasks for TransDecoder. TransDecoder first finds long ORFs with *TransDecoder.LongOrfs*, which are output as a FASTA file of protein sequences. These sequences can are then used to search against Pfam-A for conserved domains, and the coordinates from the resulting matches mapped back relative to the original transcripts. *TransDecoder.Predict* the builds the final gene models based on the training data provided by *TransDecoder.LongOrfs*, optionally using the Pfam-A results to keep ORFs which otherwise don't fit the model closely enough. Once again, note that a proper dammit config dictionary is required.

dammit.annotate.**register_user_db_tasks**(*handler*, *config*, *databases*)
Run conditional recipricol best hits LAST (CRBL) against the user-supplied databases.

dammit.annotate.**run_annotation**(*handler*)
Run the annotation pipeline from the given handler.

Prints the appropriate output and exits of the pipeline is alredy completed.

Parameters **handler** (handler.TaskHandler) – Handler with tasks for the pipeline.

2.6.4 dammit.app module

class dammit.app.**DammitApp**(*arg_src=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Bases: object

description()

epilog()

get_parser()
Build the main parser.

handle_annotate()

handle_databases()

handle_migrate()

run()

2.6.5 dammit.databases module

dammit.databases.**build_default_pipeline**(*handler*, *config*, *databases*, *with_uniref=False*, *with_nr=False*)
Register tasks for dammit's builtin database prep pipeline.

Parameters

- **handler** (handler.TaskHandler) – The task handler to register on.
- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.

- **databases** (*dict*) – The dictionary of files from *databases.json*.
- **with_uniref** (*bool*) – If True, download and install the uniref90 database. Note that this will take 16+Gb of RAM and a looong time to prepare with *lastdb*.

Returns The handler passed in.

Return type *handler.TaskHandler*

dammit.databases.**build_quick_pipeline**(*handler*, *config*, *databases*)

dammit.databases.**check_or_fail**(*handler*)
Check that the handler's tasks are complete, and if not, exit with status 2.

dammit.databases.**default_database_dir**(*logger*)
Get the default database directory: checks the environment for a DAMMIT_DB_DIR variable, and if it is not found, returns the default location of *$HOME/.dammit/databases*.

Parameters **logger** (*logging.logger*) – Logger to write to.

Returns Path to the database directory.

Return type str

dammit.databases.**get_handler**(*config*)
Build the TaskHandler for the database prep pipeline. The handler will not have registered tasks when returned.

Parameters

- **config** (*dict*) – Config dictionary, which contains the command line arguments and the entries from the config file.
- **databases** (*dict*) – The database dictionary from *databases.json*.

Returns A constructed TaskHandler.

Return type *handler.TaskHandler*

dammit.databases.**install**(*handler*)
Run the database prep pipeline from the given handler.

dammit.databases.**print_meta**(*handler*)
Print metadata about the database pipeline.

Parameters **handler** (handler.TaskHandler) – The database task handler.

dammit.databases.**register_busco_tasks**(*handler*, *config*, *databases*)

dammit.databases.**register_nr_tasks**(*handler*, *params*, *databases*)

dammit.databases.**register_orthodb_tasks**(*handler*, *params*, *databases*)

dammit.databases.**register_pfam_tasks**(*handler*, *params*, *databases*)

dammit.databases.**register_rfam_tasks**(*handler*, *params*, *databases*)

dammit.databases.**register_uniref90_tasks**(*handler*, *params*, *databases*)

2.6.6 dammit.handler module

class dammit.handler.**TaskHandler**(*directory*, *logger*, *files=None*, *profile=False*, *db=None*, *n_threads=1*, ***doit_config_kwds*)
Bases: doit.cmd_base.TaskLoader

check_uptodate()
Check if all tasks are up-to-date, ie if the pipeline is complete. Note that this moves to the handler's directory to lessen issues with relative versus absolute paths.

Returns True if all are up to date.

Return type bool

clear_tasks()
Empty the task dictionary.

get_status(*task*, *move=False*)
Get the up-to-date status of a single task.

Parameters

- **task** (*str*) – The task name to look up.
- **move** (*bool*) – If True, move to the handler's directory before checking. Whether this is necessary depends mostly on whether the task uses relative or absolute paths.

Returns The string represenation of the status. Either "run" or "uptodate".

Return type str

load_tasks(*cmd*, *opt_values*, *pos_args*)
Internal to doit – triggered by the TaskLoader.

print_statuses(*uptodate_msg='All tasks up-to-date!'*, *outofdate_msg='Some tasks out of date!'*)
Print the up-to-date status of all tasks.

Parameters

- **uptodate_msg** (*str*) – The message to print if all tasks are up to
- **date.** –

Returns A bool (True if all up to date) and a dictionary of statuses.

Return type tuple

register_task(*name*, *task*, *files=None*)
Register a new task and its files with the handler.

It may seem redundant or confusing to give the tasks a name different than their internal doit name. I do this because doit tasks need to have names as unique as possible, so that they can be reused in different projects. A particular TaskHandler instance is only used for one pipeline run, and allowing different names makes it easier to reference tasks from elsewhere.

Parameters

- **name** (*str*) – Name of the task. Does not have to correspond to doit's internal task name.
- **(** (*task*) – obj:): Either a dictionary or Task object.
- **files** (*dict*) – Dictionary of files used.

run(*doit_args=None*, *verbose=True*)
Run the pipeline. Movees to the directory, loads the tasks into doit, and executes that tasks that are not up-to-date.

Parameters

- **doit_args** (*list*) – Args that would be passed to the doit shell command. By default, just run.
- **verbose** (*bool*) – If True, print UI stuff.

Returns Exit status of the doit command.

Return type int

2.6.7 dammit.log module

dammit.log.**init_default_logger**()

dammit.log.**start_logging**(*filename=None*, *test=False*)

2.6.8 dammit.meta module

Program metadata: the version, install path, description, and default config.

dammit.meta.**get_config**()
Parse the default JSON config files and return them as dictionaries.

Returns The config and databases dictionaries.

Return type tuple

2.6.9 dammit.parallel module

dammit.parallel.**check_parallel**(*logger=None*)

dammit.parallel.**parallel_fasta**(*input_filename*, *output_filename*, *command*, *n_jobs*, *sshloginfile=None*, *check_dep=True*, *logger=None*)
Given an input FASTA source, target, shell command, and number of jobs, construct a gnu-parallel command to act on the sequences.

Parameters

- **input_filename** (*str*) – The source FASTA.
- **output_filename** (*str*) – The target.
- **command** (*list*) – The shell command (in subprocess format).
- **n_jobs** (*int*) – Number of cores or nodes to split to.
- **sshloginfile** (*str*) – Path to file with node addresses.
- **check_dep** (*bool*) – If True, check for the gnu-parallel executable.
- **logger** (*logging.Logger*) – A logger to use.

Returns The constructed shell command.

Return type str

2.6.10 dammit.profile module

class dammit.profile.**Profiler**
Bases: object

Thread-safe performance profiler.

start_profiler(*filename=None*, *blockname='__main__'*)
Start the profiler, with results stored in the given filename.

Parameters

- **filename** (*str*) – Path to store profiling results. If not given, uses a representation of the current time
- **blockname** (*str*) – Name assigned to the main block.

stop_profiler()
Shut down the profiler and write the final elapsed time.

write_result(*task_name*, *start_time*, *end_time*, *elapsed_time*)
Write results to the file, using the given task name as the name for the results block.

Parameters

- **task_name** (*str*) – ID for the result row (the block profiled).
- **start_time** (*float*) – Time of block start.
- **end_time** (*float*) – Time of block end.
- **elapsed_time** (*float*) – Total time.

dammit.profile.**StartProfiler**(*filename=None*, *blockname='__main__'*)

class dammit.profile.**Timer**
Bases: object

Simple timer class.

start()
Start the timer.

stop()
Stop the timer and return the elapsed time.

dammit.profile.**add_profile_actions**(*task*)

dammit.profile.**profile_task**(*task_func*)

dammit.profile.**setup_profiler**()
Returns a context manager, a funnction to add profiling actions to doit tasks, and a decoratator to apply that function to task functions.

The profiling function adds new actions to the beginning and end of the given task's action list, which start and stop the profiler and record the results. The task decorator applies this function. The actions only record data if the profiler is running when they are called, and they are removed from doit's execution output to reduce clutter.

The context manager starts the profiler in its block, storing data in the given file.

Yes, this is a function function function which creates six different functions at seven different function scopes. Written in honor of javascript programmers everywhere, and to baffle and irritate @ryneches.

dammit.profile.**title_without_profile_actions**(*task*)
Generate title without profiling actions

2.6.11 dammit.ui module

class dammit.ui.**GithubMarkdownReporter**(*outstream*, *options*)
Bases: doit.reporter.ConsoleReporter

Specialized doit reporter to make task output Github Markdown compliant.

execute_task(*task*)
called when excution starts

skip_ignore(*task*)
skipped ignored task

skip_uptodate(*task*)
skipped up-to-date task

dammit.ui.**checkbox**(*msg*, *checked=False*)
Generate a Github markdown checkbox for the message.

dammit.ui.**header**(*msg*, *level=1*)
Standardize output headers for submodules.

This doesn't need to be logged, but it's nice for the user.

dammit.ui.**listing**(*d*)
Generate a markdown list.

dammit.ui.**paragraph**(*msg*, *wrap=80*)
Generate a wrapped paragraph.

2.6.12 dammit.utils module

class dammit.utils.**DammitTask**(*name*, *actions*, *file_dep=()*, *targets=()*, *task_dep=()*, *uptodate=()*, *calc_dep=()*, *setup=()*, *clean=()*, *teardown=()*, *subtask_of=None*, *has_subtask=False*, *doc=None*, *params=()*, *pos_arg=None*, *verbosity=None*, *title=None*, *getargs=None*, *watch=()*, *loader=None*)
Bases: doit.task.Task

Subclass doit.task.Task for dammit. Updates the string __repr__ and adds a uniform updated title function.

title()
String representation on output.

@return: (str) Task name and actions

class dammit.utils.**Move**(*target*, *create=False*, *verbose=False*)
Bases: object

Context manager to change current working directory.

dammit.utils.**cleaned_actions**(*actions*)
Get a cleanup list of actions: Python actions have their <locals> portion stripped, which clutters up PythonActions that are closures.

dammit.utils.**dict_to_task**(*task_dict*)
Given a doit task dict, return a DammitTask.

Parameters **task_dict** (*dict*) – A doit task dict.

Returns Subclassed doit task.

Return type *DammitTask*

dammit.utils.**doit_task**(*task_dict_func*)
Wrapper to decorate functions returning pydoit Task dictionaries and have them return pydoit Task objects

dammit.utils.**touch**(*filename*)
Perform the equivalent of bash's touch on the file.

Parameters **filename** (*str*) – File path to touch.

dammit.utils.**which**(*program*)

Checks whether the given program (or program path) is valid and executable.

NOTE: Sometimes copypasta is okay! This function came from stackoverflow:

http://stackoverflow.com/a/377028/5109965

Parameters **program** (*str*) – Either a program name or full path to a program.

Returns Return the path to the executable or None if not found

2.6.13 Module contents

Python Module Index

d

Index

S

T

V

W