Fyrd Documentation

Release 0.6.1-beta.8

Mike Dacre <mike.dacre@gmail.com>

Aug 02, 2017

Contents

1 Getting Started **3**
- 1.1 Simple Job Submission . . . 3
- 1.2 Functions . . . 3
  - 1.2.1 Possible Infinate Recursion Error . . . 4
- 1.3 File Submission . . . 4
- 1.4 Keywords . . . 4
- 1.5 Profiles . . . 5

2 Configuration **7**

3 Keyword Arguments **9**
- 3.1 Adding your own keywords . . . 9

4 Console Scripts **11**
- 4.1 fyrd . . . 11
  - 4.1.1 Examples . . . 11
  - 4.1.2 All Options . . . 12
- 4.2 aliases . . . 12

5 Advanced Usage **13**
- 5.1 The Job Class . . . 13
  - 5.1.1 Script File Handling . . . 13
  - 5.1.2 Job Output Handling and Retrieval . . . 14
- 5.2 Job Files . . . 14
- 5.3 Helpers . . . 14
  - 5.3.1 Pandas . . . 15
  - 5.3.2 Running on a split file . . . 15
- 5.4 Queue Management . . . 16
- 5.5 Config . . . 17
- 5.6 Logging . . . 17

6 API Reference **19**
- 6.1 fyrd.queue . . . 19
  - 6.1.1 fyrd.queue.Queue . . . 19
  - 6.1.2 fyrd.queue functions . . . 20
- 6.2 fyrd.job . . . 21
  - 6.2.1 fyrd.job.Job . . . 22

6.3 fyrd.submission_scripts 25
6.4 fyrd.options 26
6.5 fyrd.conf 28
6.5.1 config 28
6.5.2 profiles 29
6.6 fyrd.helpers 30
6.7 fyrd.basic 33
6.8 fyrd.local 36
6.8.1 fyrd.local.JobQueue 36
6.8.2 fyrd.local.job_runner 37
6.9 fyrd.run 37
6.10 fyrd.logme 41

Python Module Index **45**

Index **47**

Python job submission on torque and slurm clusters with dependency tracking. Pronounced 'feared' (sort of), Anglo-Saxon for an army, particularly an army of freemen (an army of nodes). The logo is based on a Saxon shield commonly used in these fyrds. This library used to be known as 'Python Cluster'.

Allows simple job submission with *dependency tracking and queue waiting* on either torque, slurm, or locally with the multiprocessing module. It uses simple techniques to avoid overwhelming the queue and to catch bugs on the fly.

It is routinely tested on Mac OS and Linux with slurm and torque clusters, or in the absence of a cluster, on Python versions 2.7.10, 2.7.11, 2.7.12, 3.3.0, 3.4.0, 3.5.2, 3.6.2, and 3.7-dev. The full test suite is available in the *tests* folder.

For complete documentation see the documentation site and the Fyrd.pdf document in this repository.

Contents:

CHAPTER 1

Getting Started

1.1 Simple Job Submission

At its simplest, this module can be used by just executing *submit(<command>)*, where command is a function or system command/shell script. The module will autodetect the cluster, generate an intuitive name, run the job, and write all outputs to files in the current directory. These can be cleaned with *clean_dir()*.

To run with dependency tracking, run:

```python
import fyrd
job  = fyrd.submit(<command1>)
job2 = fyrd.submit(<command2>, depends=job1)
out  = job2.get()  # Will block until job completes
```

The *submit()* function is actually just a wrapper for the Job class. The same behavior as above can be obtained by initializing a *Job* object directly:

```python
import fyrd
job  = fyrd.Job(<command1>)
job.submit()
job2 = fyrd.Job(<command2>, depends=job1).submit()
out  = job2.get()  # Will block until job completes
```

Note that as shown above, the submit method returns the *Job* object, so it can be called on job initialization. Also note that the object returned by calling the *submit()* function (as in the first example) is also a *Job* object, so these two examples can be used fully interchangeably.

1.2 Functions

The submit function works well with python functions as well as with shell scripts and shell commands, in fact, this is the most powerful feature of this package. For example:

```python
import fyrd
def raise_me(something, power=2):
    return something**power
outs = []
if __name__ == '__main__':
    for i in range(80):
        outs.append(fyrd.submit(my_function, (i,), {'power': 2},
                                mem='10MB', time='00:00:30'))
    final_sum = 0
    for i in outs:
        final_sum += i.get()
    print(final_sum)
```

By default this will submit every instance as a job on the cluster, then get the results and clean up all intermediate files, and the code will work identically on a Mac with no cluster access, a slurm cluster, or a torque cluster, with no need to change syntax.

This is very powerful when combined with simple methods that split files or large python classes, to make this kind of work easier, a number of simple functions are provided in the helpers module, to learn more about that, review the Advanced Usage section of this documentation.

Function submission works equally well for submitting methods, however the original class object will not be updated, the method return value will be accurate, but any changes the method makes to *self* will not be returned from the cluster and will be lost.

1.2.1 Possible Infinate Recursion Error

Warning: in order for function submission to work, *fyrd* ends up importing your original script file on the nodes. This means that all code in your file will be executed, so anything that isn't a function or class must be protected with an *if __name__ == '__main__':* protecting statement.

If you do not do this you can end up with multi-submission and infinite recursion, which could mess up your jobs or just crash the job, but either way, it won't be good.

This isn't true when submitting from an interactive session such as ipython or jupyter.

1.3 File Submission

If you want to just submit a job file that has already been created, either by this software or any other method, that can be done like this:

```python
from fyrd import submit_file
submit_file('/path/to/script', dependencies=[7, 9])
```

This will return the job number and will enter the job into the queue as dependant on jobs 7 and 9. The dependencies can be omitted.

1.4 Keywords

The *Job* class, and therefore every submission script, accepts a large number of keyword arguments and synonyms to make job submission easy. Some good examples:

- cores

- mem (or memory)
- time (or walltime)
- partition (or queue)

The synonyms are provided to make submission easy for anyone familiar with the arguments used by either torque or slurm. For example:

```
job = Job('zcat huge_file | parse_file', cores=1, mem='30GB', time='24:00:00')
job = Job(my_parallel_function, cores=28, mem=12000, queue='high_mem')
for i in huge_list:
    out.append(submit(parser_function, i, cores=1, mem='1GB', partition='small'))
job = Job('ls /etc')
```

As you can see, optional keywords make submission very easy and flexible. The whole point of this software it to make working with a remote cluster in python as easy as possible.

For a full list of keyword arguments see the Keyword Arguments section of the documentation.

All options are defined in the fyrd.options module. If you want extra options, just submit an issue or add them yourself and send me a pull request.

1.5 Profiles

One of the issues with using keyword options is the nuisance of having to type them every time. More importantly, when writing code to work on any cluster one has to deal with heterogeneity between the clusters, such as the number of cores available on each node, or the name of the submission queue.

Because of this, *fyrd* makes use of profiles that bundle keyword arguments and give them a name, so that cluster submission can look like this:

```
job = Job('zcat huge_file | parse_file', profile='large')
job = Job(my_parallel_function, cores=28, profile='high_mem')
```

These profiles are defined in *~/.fyrd/profiles.txt* by default and have the following syntax:

```
[large]
partition = normal
cores = 16
nodes = 1
time = 24:00:00
mem = 32000
```

This means that you can now do this:

```
Job(my_function, profile='large')
```

You can create as many of these as you like.

While you can edit the profile file directly to add and edit profile, it is easier and more stable to use the console script:

..code:: shell

> fyrd profile list fyrd profile edit large time:02-00:00:00 mem=64GB fyrd profile edit DEFAULT partition:normal fyrd profile remove-option DEFAULT cores fyrd profile add silly cores:92 mem:1MB fyrd profile delete silly

The advantage of using the console script is that argument parsing is done on editing the profiles, so any errors are caught at that time. If you edit the file manually, then any mistakes will cause an Exception to be raised when you try to submit a job.

If no arguments are given the default profile (called 'DEFAULT' in the config file) is used.

Note: any arguments in the DEFAULT profile are available in all profiles if the are not manually overridden there. The DEFAULT profile cannot be deleted. It is a good place to put the name of the default queue.

CHAPTER 2

Configuration

Many program parameters can be set in the config file, found by default at *~/.fyrd/config.txt*.

This file has three sections with the following defaults:

[queue]:

```
max_jobs (int):     sets the maximum number of running jobs before
                    submission will pause and wait for the queue to empty
sleep_len (int):    sets the amount of time the program will wait between
                    submission attempts
queue_update (int): sets the amount of time between refreshes of the queue.
res_time (int):     Time in seconds to wait if a job is in an uncertain
                    state, usually preempted or suspended. These jobs often
                    resolve into running or completed again after some time
                    so it makes sense to wait a bit, but not forever. The
                    default is 45 minutes: 2700 seconds.
queue_type (str):   the type of queue to use, one of 'torque', 'slurm',
                    'local', 'auto'. Default is auto to auto-detect the
                    queue.
```

[jobs]:

```
clean_files (bool):    means that by default files will be deleted when job
                       completes
clean_outputs (bool):  is the same but for output files (they are saved
                       first)
file_block_time (int): Max amount of time to block after job completes in
                       the queue while waiting for output files to appear.
                       Some queues can take a long time to copy files under
                       load, so it is worth setting this high, it won't
                       block unless the files do not appear.
filepath (str):        Path to write all temp and output files by default,
                       must be globally cluster accessible. Note: this is
                       *not* the runtime path, just where files are written
                       to.
suffix (str):          The suffix to use when writing scripts and output
```

```
                           files
auto_submit (bool):        If wait() or get() are called prior to submission,
                           auto-submit the job. Otherwise throws an error and
                           returns None
generic_python (bool): Use /usr/bin/env python instead of the current
                           executable, not advised, but sometimes necessary.
profile_file (str):        the config file where profiles are defined.
```

[jobqueue]:

```
Sets options for the local queue system, will be removed in the future in
favor of database.

jobno (int):  The current job number for the local queue, auto-increments
              with every submission.
```

Example file:

```
[queue]
res_time = 2700
queue_type = auto
sleep_len = 1
queue_update = 2
max_jobs = 1000
bool = True

[jobs]
suffix = cluster
file_block_time = 12
filepath = None
clean_outputs = False
auto_submit = True
profile_file = /Users/dacre/.fyrd/profiles.txt
clean_files = True
generic_python = False

[jobqueue]
jobno = 9
```

The config is managed by fyrd/conf.py and enforces a minimum set of entries. If the config does not exist or any entries are missing, they will be created on the fly using the defaults defined in the defaults.

CHAPTER 3

Keyword Arguments

To make submission easier, this module defines a number of keyword arguments in the options.py file that can be used for all submission and Job() functions. These include things like 'cores' and 'nodes' and 'mem'.

The following is a complete list of arguments that can be used in this version

> depends clean_files clean_outputs cores modules syspaths scriptpath outpath runpath suffix outfile errfile imports threads nodes features qos time mem partition account export begin

Note: Type is enforced, any provided argument must match that python type (automatic conversion is attempted), the default is just a recommendation and is not currently used. These arguments are passed like regular arguments to the submission and Job() functions, eg:

```
Job(nodes=1, cores=4, mem='20MB')
```

This will be interpretted correctly on any system. If torque or slurm are not available, any cluster arguments will be ignored. The module will attempt to honor the cores request, but if it exceeds the maximum number of cores on the local machine, then the request will be trimmed accordingly (i.e. a 50 core request will become 8 cores on an 8 core machine).

3.1 Adding your own keywords

There are many more options available for torque and slurm, to add your own, edit the options.py file, and look for CLUSTER_OPTS (or TORQUE/SLURM if your keyword option is only availble on one system). Add your option using the same format as is present in that file. The format is:

```
('name', {'slurm': '--option-str={}', 'torque': '--torque-option={}',
          'help': 'This is an option!', 'type': str, 'default': None})
```

You can also add list options, but they must include 'sjoin' and 'tjoin' keys to define how to merge the list for slurm and torque, or you must write custom option handling code in `fyrd.options.options_to_string()`. For an excellent example of both approaches included in a single option, see the 'features' keyword above.

CHAPTER 4

Console Scripts

This software is primarily intended to be a library, however some management tasks are just easier from the console. For that reason, *fyrd* has a frontend console script that makes tasks such as managing the local config and profiles trivial, it also has modes to inspect the queue easily, and to wait for jobs from the console, as well as to clean the working directory.

4.1 fyrd

This software has uses a subcommand system to separate modes, and has six modes:

- *config* — show and edit the contents of the config file
- *profile* - inspect and manage cluster profiles
- *keywords* - print a list of current keyword arguments with descriptions for each
- *queue* - show running jobs, makes filtering jobs very easy
- *wait* - wait for a list of jobs
- *clean* - clean all script and output files in the given directory

Several of the commands have aliases (*conf* and *prof* being the two main ones)

4.1.1 Examples

```
fyrd prof list
fyrd prof add large cores:92 mem:200GB partition:high_mem time:00:06:00
```

```
fyrd queue  # Shows all of your current jobs
fyrd queue -a # Shows all users jobs
fyrd queue -p long -u bob dylan # Show all jobs owned by bob and dylan in the long␣
↪queue
```

```
fyrd wait 19872 19876
fyrd wait -u john
```

```
fyrd clean
```

4.1.2 All Options

4.2 aliases

Several shell scripts are provided in *bin/* to provide shortcuts to the *fyrd* subcommands:

- *my-queue* (or *myq*): *fyrd queue*
- *clean-job-files*: *fyrd clean*
- *monitor-jobs*: *fyrd wait*
- *cluster-keywords*: *fyrd keywords*

CHAPTER 5

Advanced Usage

Most of the important functionality is covered in the Getting Started section, and full details on the library are available in the API Reference section. This section just provides some extra information on Job and Queue management, and importantly introduces some of the higher-level options available through the helpers.

5.1 The Job Class

The core of this submission system is the *Job* class, this class builds a job using keyword arguments and profile parsing. The bulk of this is done at class initialization and is covered in the getting started section of this documentation and on job submission with the *submit()* method. There are several other features of this class to be aware of though.

5.1.1 Script File Handling

Torque and slurm both require submission scripts to work. In the future these will be stored by fyrd in a database and submitted from memory, but for now they are written to disk.

The creation and writing of these scripts is handled by the Script and Function classes in the fyrd.submission_scripts module. These classes pass keywords to the options_to_string() function of the options method, which converts them into a submission string compatible with the active cluster. These are then written to a script for submission to the cluster.

The *Function* class has some additional functionality to allow easy submission of functions to the cluster. It tries to build a list of all possible modules that the function could need and adds import statements to all of them to the function submission script. It then pickles the submitted function and arguments to a pickle file on the disk, and writes a python script to the same directory.

This python script unpickles the function and arguments and runs them, pickling either the result or and exception, if one is raised, to the disc on completion. The submission script calls this python script on the cluster nodes.

The script and output files are written to the path defined by the *.filepath* attribute of the *Job* class, which is set using the ‘filepath’ keyword argument. If not set, this directory defaults to the directory set in the filepath section of the config file or the current working directory. Note that this path is independent of the *.runpath* attibute, which is where the code will actually run, and also defaults to the current working directory.

5.1.2 Job Output Handling and Retrieval

The correct way to get outputs from within a python session is to call the *.get()* method of the *Job* class. This first calls the *.wait()* method, which blocks until job completion, and then the *.fetch_outputs()* method which *calls get_output, get_stdout, and get_stderr, which save all function outputs, STDOUT, and STDERR to the class*. This means that outputs can be accessed using the following *Job* class attributes:

- *.output* — the function output for functions or STDOUT for scripts
- *.stdout* — the STDOUT for the script submission (always present)
- *.stderr* — the STDERR for the script submission (always present)

This makes job output retrieval very easy, but it is sometimes not what you want, particularly if outputs are very large (they get loaded into memory).

The *wait()* method will not save any outputs. In addition *get()* can be with the *save=False* argument, which means it will fetch the output (or STDOUT) only, but will not write them to the class itself.

Note: By default, *get()* also deletes all script and output files. This is generally a good thing as it keeps the working directory clean, but it isn't always what you want. To prevent outputs from being deleted, pass *delete_outfiles=False* to *get()*, or alternatively set the *.clean_outputs* attribute to *False* prior to running *get()*. To prevent the cleaning of any files, including the script files, pass *cleanup=False* or set *.clean_files* to *False*.

clean_files and *clean_outputs* can also be set globally in the config file.

5.2 Job Files

All jobs write out a job file before submission, even though this is not necessary (or useful) with multiprocessing. This will change in a future version.

To ensure files are obviously produced by this package and that files are unique the file format is name.number.random_string.suffix.extension. These are:

name: Defined by the *name*= argument or guessed from the function/script number: A number count of the total jobs with the same name already queued random_string: An 8-character random string suffix: A string defined in the config file, default 'cluster' extension: An obvious extension such as '.sbatch' or '.qsub'

To change the directory these files are written to, set the filedir item in the config file or use the 'filedir' keyword argument to Job or submit.

NOTE: This directory *must* be accessible to the compute nodes!!!

It is sometimes useful to set the filedir setting in the config to a single directory accessible cluster-wide. This avoids cluttering the current directory, particularly as outputs can be retrieved so easily from within python. If you are going to do this set the 'clean_files' and 'clean_outfiles' arguments in the config file to avoid cluttering the directory.

All Job objects have a `clean()` method that will delete any left over files. In addition there is a clean_job_files script that will delete all files made by this package in any given directory. Be very careful with the script though, it can clobber a lot of work all at once if it is used wrong.

5.3 Helpers

The fyrd.helpers module defines several simple functions that allow more complex job handling.

The helpers are all high level functions that are not required for the library but make difficult jobs easy to assist in the goal of trivially easy cluster submission.

5.3.1 Pandas

The most important function in *fyrd.helpers* is *parapply()*, which allows the user to submit a *pandas.DataFrame.apply* method to the cluster in parallel by splitting the DataFrame, submitting jobs, and then recombining the DataFrame at the end, all without leaving any temp files behind. e.g.:

```
df = pandas.read_csv('my_huge_file.txt')
df = fyrd.helpers.parapply(100, df, long_running_function, profile='fast')
```

That command will split the dataframe into 100 pieces, submit each to the cluster as a different job with the profile 'fast', and then recombine them into a single DataFrame again at the end.

parapply_summary behaves similarly but assumes that the function summarizes the data rather than returning a DataFrame of the same size. It thus runs the function on the resulting DataFrame also, allowing all dfs to be merged. e.g.:

```
df = fyrd.helpers.parapply_summary(df, numpy.mean)
```

This will return just the mean of all the numeric columns, *parapply* would return a DataFrame with duplicates for every submitted job.

5.3.2 Running on a split file

The splitrun function behaves similarly to the *parapply()* function, with the exception that it works on a filesystem file instead, which it splits into pieces. It then runs your job on all of the pieces and attempts to recombine them, deleting the intermediate files as it goes.

If you specify an output file, the outputs are merged and places into that file, otherwise, if the outputs are a string (always true for scripts), the function returns a merged string. If the outputs are not strings, then the function just returns a list out outputs that you will have to combine yourself.

The key to this function is that if the job is a script, it must at a minimum contain '{file}' where the file argument goes, and if the job is a function it must contain and argument or keyword argument that matches '<file>'.

If you expect the job to have and output, you must provide the *outfile=* argument too, and be sure that '{outfile}' is present in the script, if a script, or '<outfile>' is in either args or kwargs if a function.

In addition, you should pass *inheader=True* if the input file has a header line, and *outheader=True* if the same is true for the outfile. It is very important to pass these arguments, because they both will strip the top line from a file if True. Importantly, if *inheader* is *True* on a file without a header, the top line will appear at the top of every broken up file.

Examples:

```
script = """my_long_script --in {file} --out {outfile}"""
outfile = fyrd.helpers.splitrun(
    100, 'huge_file.txt', script, name='my_job', profile='long',
    outfile='output.txt', inheader=True, outheader=True
)
```

```
output = fyrd.helpers.splitrun(
    100, 'huge_file.txt', function, args=('<file>',), name='my_job',
    profile='long', outfile='output.txt', inheader=True, outheader=True
)
```

5.4 Queue Management

Queue handling is done by the Queue class in the fyrd.queue module. This class calls the fyrd.queue.queue_parser iterator which in turn calls either fyrd.queue.torque_queue_parser or fyrd.queue.slurm_queue_parser depending on the detected cluster environment (set by *fyrd.queue.QUEUE_MODE* and overridden by the 'queue_type' config option if desired (not necessary, queue type is auto-detected)).

These iterators return the following information from the queue:

```
job_id, name, userid, partition, state, node-list, node-count, cpu-per-node, exit-code
```

These pieces of information are used to create QueueJob objects for every job, which are stored in the *Queue.jobs* attribute (a dictionary). The *Queue* class provides several properties, attributes, and methods to allow easy filtering of these jobs.

Most important is the *QueueJob.state* attribute, which holds information on the current state of that job. To get a list of all states in the queue, call the *Queue.job_states* property, which will return a list of states in the queue. All of these states are also attributes of the *Queue* class, for example:

```
fyrd.Queue.completed
```

returns all completed jobs in the queue as a dictionary (a filtered copy of the *.jobs* attribute).

Note: torque states are auto-converted to slurm states, as slurm states are easier to read. e.g. 'C' becomes 'completed'.

The most useful method of *Queue* is *wait()*, it will take a list of job numbers or *Job* objects and wait until all of them are complete. This method is called by the *Job.wait()* method, and can be called directly to wait for an arbitrary number of jobs.

To wait for all jobs from a given user, you can do this:

```
q = fyrd.Queue()
q.wait(q.get_user_jobs(['bob', 'fred']))
```

This task can also be accomplished with the console application:

```
fyrd wait <job> [<job>...]
fyrd wait -u bob fred
```

The method can actually be simply accessed as a function instead of needing the *Queue* class:

```
fyrd.wait([1,2,3])
```

To generate a *Queue* object, do the following:

```
import fyrd
q = fyrd.Queue(user='self')
```

This will give you a simple queue object containg a list of jobs that belong to you. If you do not provide user, all jobs are included for all users. You can provide *qtype* to explicitly force the queue object to contain jobs from one queing system (e.g. local or torque).

To get a dictionary of all jobs, running jobs, queued jobs, and complete jobs, use:

```
q.jobs
q.running
q.complete
q.queued
```

Every job is a *QueueJob* class and has a number of attributes, including owner, nodes, cores, memory.

5.5 Config

Many of the important options used by this software are set in a config file and can be managed on the console with *fyrd conf ...*.

For full information see the Configuration section of this documentation.

5.6 Logging

I use a custion logging script called logme to log errors. To get verbose output, set *fyrd.logme.MIN_LEVEL* to 'debug' or 'verbose'. To reduce output, set logme.MIN_LEVEL to 'warn'.

CHAPTER 6

API Reference

6.1 fyrd.queue

The core class in this file is the *Queue()* class which does most of the queue management. In addition, *get_cluster_environment()* attempts to autodetect the cluster type (*torque*, *slurm*, *normal*) and sets the global cluster type for the whole file. Finally, the *wait()* function accepts a list of jobs and will block until those jobs are complete.

The Queue class relies on a few simple queue parsers defined by the *torque_queue_parser* and *slurm_queue_parser* functions. These call *qstat -x* or *squeue* and *sacct* to get job information, and yield a simple tuple of that data with the following members:

```
job_id, name, userid, partition, state, node-list, node-count, cpu-per-node, exit-code
```

The Queue class then converts this information into a *Queue.QueueJob* object and adds it to the internal *jobs* dictionary within the Queue class. This list is now the basis for all of the other functionality encoded by the Queue class. It can be accessed directly, or sliced by accessing the *completed*, *queued*, and *running* attributes of the Queue class, these are used to simply divide up the jobs dictionary to make finding information easy.

6.1.1 fyrd.queue.Queue

class `fyrd.queue.`**`Queue`**(*user=None*, *partition=None*, *qtype=None*)

Bases: `object`

A wrapper for torque, slurm, or local queues.

`jobs`
dict – {jobid: Queue.QueueJob}

`max_jobs`
int – The maximum number of jobs allowed in the queue

`job_states`
int – A list of the different states of jobs in this queue

`active_job_count`
int – A count of all jobs that are either queued or running in the current queue

`can_submit`
bool – True if total active jobs is less than max_jobs

Can filter by user, queue type or partition on initialization.

Parameters

- **`user`** (*`str`*) – Optional usernameto filter the queue with. If user='self' or 'current', the current user will be used.
- **`partition`** (*`str`*) – Optional partition to filter the queue with.
- **`qtype`** (*`str`*) – 'torque', 'slurm', or 'local', defaults to auto-detect.

Methods

`Queue.`**`wait`**(*jobs*)
Block until all jobs in jobs are complete.

Update time is dependant upon the queue_update parameter in your ~/.fyrd file.

In addition, wait() will not return until between 1 and 3 seconds after a job has completed, irrespective of queue_update time. This allows time for any copy operations to complete after the job exits.

Parameters **`jobs`** – A job or list of jobs to check. Can be one of: Job or multiprocessing.pool.ApplyResult objects, job ID (int/str), or a object or a list/tuple of multiple Jobs or job IDs.

Returns True on success False or None on failure.

`Queue.`**`wait_to_submit`**(*max_jobs=None*)
Block until fewer running/queued jobs in queue than max_jobs.

Parameters **`max_jobs`** (*`int`*) – Override self.max_jobs

`Queue.`**`get_jobs`**(*key*)
Return a dict of jobs where state matches key.

`Queue.`**`update`**()
Refresh the list of jobs from the server, limit queries.

exception `fyrd.queue.`**`QueueError`**
Simple Exception wrapper.

6.1.2 fyrd.queue functions

parsers

`fyrd.queue.`**`queue_parser`**(*qtype=None*, *user=None*, *partition=None*)
Call either torque or slurm qtype parsers depending on qtype.

Parameters

- **`qtype`** – Either 'torque' or 'slurm', defaults to current MODE
- **`user`** – optional user name to pass to queue to filter queue with

Yields *tuple* –

job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

`fyrd.queue.`**`torque_queue_parser`**(*user=None*, *partition=None*)

Iterator for torque queues.

Use the *qstat -x* command to get an XML queue for compatibility.

Parameters

- **`user`** – optional user name to pass to qstat to filter queue with
- **`partiton`** – optional partition to filter the queue with

Yields *tuple* –

job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

numcpus is currently always 1 as most torque queues treat every core as a node.

`fyrd.queue.`**`slurm_queue_parser`**(*user=None*, *partition=None*)

Iterator for slurm queues.

Use the *squeue -O* command to get standard data across implementation, supplement this data with the results of *sacct*. sacct returns data only for the current user but retains a much longer job history. Only jobs not returned by squeue are added with sacct, and they are added to *the end* of the returned queue, i.e. *out of order with respect to the actual queue*.

Parameters

- **`user`** – optional user name to filter queue with
- **`partition`** – optional partition to filter queue with

Yields *tuple* –

job_id, name, userid, partition, state, nodelist, numnodes, ntpernode, exit_code

utilities

`fyrd.queue.`**`get_cluster_environment`**()

Detect the local cluster environment and set MODE globally.

Uses which to search for sbatch first, then qsub. If neither is found, MODE is set to local.

Returns MODE variable ('torque', 'slurm', or 'local')

Return type tuple

`fyrd.queue.`**`check_queue`**(*qtype=None*)

Raise exception if MODE is incorrect.

6.2 fyrd.job

Job management is handled by the *Job()* class. This is a very large class that defines all the methods required to build and submit a job to the cluster.

It accepts keyword arguments defined in *fyrd.options* on initialization, which are then fleshed out using profile information from the config files defined by *fyrd.conf*.

The primary argument on initialization is the function or script to submit.

Examples:

```
Job('ls -lah | grep myfile')
Job(print, ('hi',))
Job('echo hostname', profile='tiny')
Job(huge_function, args=(1,2) kwargs={'hi': 'there'},
    profile='long', cores=28, mem='200GB')
```

6.2.1 fyrd.job.Job

class fyrd.**Job**(*command*, *args=None*, *kwargs=None*, *name=None*, *qtype=None*, *profile=None*, ***kwds*)

Bases: `object`

Information about a single job on the cluster.

Holds information about submit time, number of cores, the job script, and more.

Below are the core attributes and methods required to use this class.

out
str – The output of the function or a copy of stdout for a script

stdout
str – Any output to STDOUT

stderr
str – Any output to STDERR

exitcode
int – The exitcode of the running processes (the script runner if the Job is a function.

start
datetime – A datetime object containing time execution started on the remote node.

end
datetime – Like start but when execution ended.

runtime
timedelta – A timedelta object containing runtime.

files
list – A list of script files associated with this class

done
bool – True if the job has completed

submit()
submit the job if it is ready

wait()
block until the job is done

get()
block until the job is done and then return the output (stdout if job is a script), by default saves all outputs to self (i.e. .out, .stdout, .stderr) and deletes all intermediate files before returning. If *save* argument is *False*, does not delete the output files by default.

clean()
delete any files created by this object

Printing or reproducing the class will display detailed job information.

Both *wait()* and *get()* will update the queue every few seconds (defined by the queue_update item in the config) and add queue information to the job as they go.

If the job disappears from the queue with no information, it will be listed as 'completed'.

All jobs have a .submission attribute, which is a Script object containing the submission script for the job and the file name, plus a 'written' bool that checks if the file exists.

In addition, SLURM jobs have a .exec_script attribute, which is a Script object containing the shell command to _run. This difference is due to the fact that some SLURM systems execute multiple lines of the submission file at the same time.

Finally, if the job command is a function, this object will also contain a *.function* attribute, which contains the script to run the function.

Initialization function arguments.

Parameters

- **`command`** (*`function/str`*) – The command or function to execute.
- **`args`** (*`tuple/dict`*) – Optional arguments to add to command, particularly useful for functions.
- **`kwargs`** (*`dict`*) – Optional keyword arguments to pass to the command, only used for functions.
- **`name`** (*`str`*) – Optional name of the job. If not defined, guessed. If a job of the same name is already queued, an integer job number (not the queue number) will be added, ie. <name>.1
- **`qtype`** (*`str`*) – Override the default queue type
- **`profile`** (*`str`*) – The name of a profile saved in the conf
- **`other keywords are parsed into cluster keywords by the`** (*`*All`*) –
- **`system. For available keywords see `fyrd.option_help()`*`** (*`options`*) –

Methods

`Job.`**`write`**(*overwrite=True*)

Write all scripts.

Parameters **`overwrite`** (*`bool`*) – Overwrite existing files, defaults to True.

`Job.`**`clean`**(*delete_outputs=None*, *get_outputs=True*)

Delete all scripts created by this module, if they were written.

Parameters

- **`delete_outputs`** (*`bool`*) – also delete all output and err files, but get their contents first.
- **`get_outputs`** (*`bool`*) – if delete_outputs, save outputs before deleting.

`Job.`**`submit`**(*wait_on_max_queue=True*)

Submit this job.

Parameters **`wait_on_max_queue`** (*`bool`*) – Block until queue limit is below the maximum before submitting.

To disable max_queue_len, set it to 0. None will allow override by the default settings in the config file, and any positive integer will be interpretted to be the maximum queue length.

Returns self

`Job.`**`wait`**`()`
Block until job completes.

`Job.`**`get`**`(`*save=True*, *cleanup=None*, *delete_outfiles=None*, *del_no_save=None*, *raise_on_error=True*`)`
Block until job completed and return output of script/function.

By default saves all outputs to this class and deletes all intermediate files.

Parameters

- **`save`** (*`bool`*) – Save all outputs to the class also (advised)
- **`cleanup`** (*`bool`*) – Clean all intermediate files after job completes.
- **`delete_outfiles`** (*`bool`*) – Clean output files after job completes.
- **`del_no_save`** (*`bool`*) – Delete output files even if *save* is *False*
- **`raise_on_error`** (*`bool`*) – If the returned output is an Exception, raise it.

Returns Function output if Function, else STDOUT

Return type str

`Job.`**`get_output`**`(`*save=True*, *delete_file=None*, *update=True*, *raise_on_error=True*`)`
Get output of function or script.

This is the same as stdout for a script, or the function output for a function.

By default, output file is kept unless delete_file is True or self.clean_files is True.

Parameters

- **`save`** (*`bool`*) – Save the output to self.out, default True. Would be a good idea to set to False if the output is huge.
- **`delete_file`** (*`bool`*) – Delete the output file when getting
- **`update`** (*`bool`*) – Update job info from queue first.
- **`raise_on_error`** (*`bool`*) – If the returned output is an Exception, raise it.

Returns The output of the script or function. Always a string if script.

`Job.`**`get_stdout`**`(`*save=True*, *delete_file=None*, *update=True*`)`
Get stdout of function or script, same for both.

By default, output file is kept unless delete_file is True or self.clean_files is True.

Parameters

- **`save`** (*`bool`*) – Save the output to self.stdout, default True. Would be a good idea to set to False if the output is huge.
- **`delete_file`** (*`bool`*) – Delete the stdout file when getting
- **`update`** (*`bool`*) – Update job info from queue first.

Returns

The contents of STDOUT, with runtime info and trailing newline removed.

Return type str

Also sets self.start and self.end from the contents of STDOUT if possible.

`Job.`**`get_stderr`**`(`*save=True*, *delete_file=None*, *update=True*`)`

Get stderr of function or script, same for both.

By default, output file is kept unless delete_file is True or self.clean_files is True.

Parameters

- **`save`** (*`bool`*) – Save the output to self.stdout, default True. Would be a good idea to set to False if the output is huge.
- **`delete_file`** (*`bool`*) – Delete the stdout file when getting
- **`update`** (*`bool`*) – Update job info from queue first.

Returns The contents of STDERR, with trailing newline removed.

Return type str

`Job.`**`get_times`**`(`*update=True*`)`

Get stdout of function or script, same for both.

Parameters **`update`** (*`bool`*) – Update job info from queue first.

Returns start, end as two datetime objects.

Return type tuple

Also sets self.start and self.end from the contents of STDOUT if possible.

`Job.`**`get_exitcode`**`(`*update=True*`)`

Try to get the exitcode.

Parameters **`update`** (*`bool`*) – Update job info from queue first.

Returns The exitcode of the running process.

Return type int

`Job.`**`update`**`(`*fetch_info=True*`)`

Update status from the queue.

Parameters **`fetch_info`** (*`bool`*) – Fetch basic job info if complete.

`Job.`**`update_queue_info`**`()`

Set queue_info from the queue even if done.

`Job.`**`fetch_outputs`**`(`*save=True*, *delete_files=None*, *get_stats=True*`)`

Save all outputs in their current state. No return value.

This method does not wait for job completion, but merely gets the outputs. To wait for job completion, use *get()* instead.

Parameters

- **`save`** (*`bool`*) – Save all outputs to the class also (advised)
- **`delete_files`** (*`bool`*) – Delete the output files when getting, only used if save is True
- **`get_stats`** (*`bool`*) – Try to get exitcode.

6.3 fyrd.submission_scripts

This module defines to classes that are used to build the actual jobs for submission, including writing the files. *Function* is actually a child class of *Script*.

class `fyrd.submission_scripts.`**`Script`**(*file_name*, *script*)

Bases: `object`

A script string plus a file name.

Initialize the script and file name.

`clean`(*delete_output=None*)

Delete any files made by us.

`exists`

True if file is on disk, False if not.

`write`(*overwrite=True*)

Write the script file.

class `fyrd.submission_scripts.`**`Function`**(*file_name*, *function*, *args=None*, *kwargs=None*, *imports=None*, *syspaths=None*, *pickle_file=None*, *outfile=None*)

Bases: `fyrd.submission_scripts.Script`

A special Script used to run a function.

Create a function wrapper.

NOTE: Function submission will fail if the parent file's code is not wrapped in an if __main__ wrapper.

Parameters

- **`file_name`** (*`str`*) – A root name to the outfiles
- **`function`** (*`callable`*) – Function handle.
- **`args`** (*`tuple`*) – Arguments to the function as a tuple.
- **`kwargs`** (*`dict`*) – Named keyword arguments to pass in the function call
- **`imports`** (*`list`*) – A list of imports, if not provided, defaults to all current imports, which may not work if you use complex imports. The list can include the import call, or just be a name, e.g ['from os import path', 'sys']
- **`syspaths`** (*`list`*) – Paths to be included in submitted function
- **`pickle_file`** (*`str`*) – The file to hold the function.
- **`outfile`** (*`str`*) – The file to hold the output.

`clean`(*delete_output=False*)

Delete the input pickle file and any scripts.

Parameters **`delete_output`** (*`bool`*) – Delete the output pickle file too.

`write`(*overwrite=True*)

Write the pickle file and call the parent Script write function.

6.4 fyrd.options

All keyword arguments are defined in dictionaries in the *options.py* file, alongside function to manage those dictionaries. Of particular importance is *option_help()*, which can display all of the keyword arguments as a string or a table. *check_arguments()* checks a dictionary to make sure that the arguments are allowed (i.e. defined), it is called on all keyword arguments in the package.

To see keywords, run *fyrd keywords* from the console or *fyrd.option_help()* from a python session.

The way that option handling works in general, is that all hard-coded keyword arguments must contain a dictionary entry for 'torque' and 'slurm', as well as a type declaration. If the type is NoneType, then the option is assumed to be a boolean option. If it has a type though, *check_argument()* attempts to cast the type and specific idiosyncrasies are handled in this step, e.g. memory is converted into an integer of MB. Once the arguments are sanitized *format()* is called on the string held in either the 'torque' or the 'slurm' values, and the formatted string is then used as an option. If the type is a list/tuple, the 'sjoin' and 'tjoin' dictionary keys must exist, and are used to handle joining.

The following two functions are used to manage this formatting step.

option_to_string() will take an option/value pair and return an appropriate string that can be used in the current queue mode. If the option is not implemented in the current mode, a debug message is printed to the console and an empty string is returned.

options_to_string() is a wrapper around *option_to_string()* and can handle a whole dictionary of arguments, it explicitly handle arguments that cannot be managed using a simple string format.

`fyrd.options.`**`option_help`**(*mode='string'*, *qtype=None*, *tablefmt='simple'*)

Print a sting to stdout displaying information on all options.

Parameters

- **`mode`** (`str`) – string: Return a formatted string print: Print the string to stdout list: Return a simple list of keywords table: Return a table of lists merged_table: Combine all keywords into a single table
- **`qtype`** (`str`) – If provided only return info on that queue type.
- **`tablefmt`** (`str`) – A tabulate-style table format, one of:

```
'plain', 'simple', 'grid', 'pipe', 'orgtbl',
'rst', 'mediawiki', 'latex', 'latex_booktabs'
```

Returns A formatted string

Return type str

`fyrd.options.`**`sanitize_arguments`**(*kwds*)

Run check_arguments, but return unmatched keywords as is.

`fyrd.options.`**`split_keywords`**(*kwargs*)

Split a dictionary of keyword arguments into two dictionaries.

The first dictionary will contain valid arguments for fyrd, the second will contain all others.

Returns (dict, dict) — valid args for fyrd, other args

Return type tuple

`fyrd.options.`**`check_arguments`**(*kwargs*)

Make sure all keywords are allowed.

Raises OptionsError on error, returns sanitized dictionary on success.

Note: Checks in SYNONYMS if argument is not recognized, raises OptionsError if it is not found there either.

`fyrd.options.`**`options_to_string`**(*option_dict*, *qtype=None*)

Return a multi-line string for slurm or torque job submission.

Parameters

- **`option_dict`** (`dict`) – Dict in format {option: value} where value can be None. If value is None, default used.
- **`qtype`** (`str`) – 'torque', 'slurm', or 'local': override queue.MODE

Returns A multi-line string of torque or slurm options.

Return type str

`fyrd.options.`**`option_to_string`**(*option*, *value=None*, *qtype=None*)

Return a string with an appropriate flag for slurm or torque.

Parameters

- **`option`** – An allowed option definied in options.all_options
- **`value`** – A value for that option if required (if None, default used)
- **`qtype`** – 'torque', 'slurm', or 'local': override queue.MODE

Returns A string with the appropriate flags for the active queue.

Return type str

6.5 fyrd.conf

fyrd.conf handles the config (*~/.fyrd/config.txt*) file and the profiles (*~/.fyrd/profiles.txt*) file.

Profiles are combinations of keyword arguments that can be called in any of the submission functions. Both the config and profiles are just ConfigParser objects, *conf.py* merely adds an abstraction layer on top of this to maintain the integrity of the files.

6.5.1 config

The config has three sections (and no defaults):

- queue — sets options for handling the queue
- jobs — sets options for submitting jobs
- jobqueue — local option handling, will be removed in the future

For a complete reference, see the config documentation : Configuration

Options can be managed with the *get_option()* and *set_option()* functions, but it is actually easier to use the console script:

```
fyrd conf list
fyrd conf edit max_jobs 3000
```

`fyrd.conf.`**`get_option`**(*section=None*, *key=None*, *default=None*)

Get a single key or section.

All args are optional, if they are missing, the parent section or entire config will be returned.

Parameters

- **`section`** (*`str`*) – The config section to use (e.g. queue), if None, all sections returned.
- **`key`** (*`str`*) – The config key to get (e.g. 'max_jobs'), if None, whole section returned.
- **`default`** – If the key does not exist, create it with this default value.

Returns Option value if key exists, None if no key exists.

`fyrd.conf.`**`set_option`**(*section*, *key*, *value*)

Write a config key to the config file.

Parameters

- **section** (*str*) – Section of the config file to use.
- **key** (*str*) – Key to add.
- **value** – Value to add for key.

Returns ConfigParser

`fyrd.conf.`**`delete`**`(`*section*, *key*`)`

Delete a config item.

Parameters

- **section** (*str*) – Section of config file.
- **key** (*str*) – Key to delete

Returns ConfigParger

`fyrd.conf.`**`load_config`**`()`

Load config from the config file.

If any section or key from DEFAULTS is not present in the config, it is added back, enforcing a minimal configuration.

Returns Config options.

Return type ConfigParser

`fyrd.conf.`**`write_config`**`()`

Write the current config to CONFIG_FILE.

`fyrd.conf.`**`create_config`**`(`*cnf=None*, *def_queue=None*`)`

Create an initial config file.

Gets all information from the file-wide DEFAULTS constant and overwrites specific keys using the values in cnf.

This means that any records in the cnf dict that are not present in DEFAULTS will be ignored, and any records that are absent will be populated from DEFAULTS.

Parameters

- **cnf** (*dict*) – A dictionary of config defaults.
- **def_queue** (*str*) – A name for a queue to add to the default profile.

`fyrd.conf.`**`create_config_interactive`**`(`*prompt=True*`)`

Interact with the user to create a new config.

Uses readline autocompletion to make setup easier.

Parameters **prompt** (*bool*) – As for confirmation before beginning wizard.

6.5.2 profiles

Profiles are wrapped in a *Profile()* class to make attribute access easy, but they are fundamentally just dictionaries of keyword arguments. They can be created with *cluster.conf.Profile(name, {keywds})* and then written to a file with the *write()* method.

The easiest way to interact with profiles is not with class but with the *get_profile()*, *set_profile()*, and *del_profile()* functions. These make it very easy to go from a dictionary of keywords to a profile.

Profiles can then be called with the *profile=* keyword in any submission function or Job class.

As with the config, profile management is the easiest and most stable when using the console script:

```
fyrd profile list
fyrd profile add very_long walltime:120:00:00
fyrd profile edit default partition:normal cores:4 mem:10GB
fyrd profile delete small
```

fyrd.conf.Profile

class fyrd.conf.**Profile**(*name*, *kwds*)

Bases: `object`

A job submission profile. Just a thin wrapper around a dict.

Set up bare minimum attributes.

Parameters

- **name** (*str*) – Name of the profile
- **kwds** (*dict*) – Dictionary of keyword arguments (will be validated).

write()

Write self to config file.

fyrd.conf.**set_profile**(*name*, *kwds*, *update=True*)

Write profile to config file.

Parameters

- **name** (*str*) – The name of the profile to add/edit.
- **kwds** (*dict*) – Keyword arguments to add to the profile.
- **update** (*bool*) – Update the profile rather than overwriting it.

fyrd.conf.**get_profile**(*profile=None*, *allow_none=True*)

Return a profile if it exists, if None, return all profiles.

Will return None if profile is supplied but does not exist.

Parameters

- **profile** (*str*) – The name of a profile to search for.
- **allow_none** (*bool*) – If True, return None if no profile matches, otherwise raise a ValueError.

Returns The requested profile.

Return type *fyrd.conf.Profile*

6.6 fyrd.helpers

The helpers are all high level functions that are not required for the library but make difficult jobs easy to assist in the goal of trivially easy cluster submission.

The functions in *fyrd.basic* below are different in that they provide simple job submission and management, while the functions in *fyrd.helpers* allow the submission of many jobs.

`fyrd.helpers.`**`parapply`**(*jobs*, *df*, *func*, *args=()*, *profile=None*, *applymap=False*, *merge_axis=0*, *merge_apply=False*, *name='parapply'*, *imports=None*, *direct=True*, ***kwds*)

Split a dataframe, run apply in parallel, return result.

This function will split a dataframe into however many pieces are requested with the jobs argument, run apply in parallel by submitting the jobs to the cluster, and then recombine the outputs.

If the 'clean_files' and 'clean_outputs' arguments are not passed, we delete all intermediate files and output files by default.

This function will take any keyword arguments accepted by Job, which can be found by running fyrd.options.option_help(). It also accepts any of the keywords accepted by by pandas.DataFrame.apply(), found here

Parameters

- **`jobs`** (*`int`*) – Number of pieces to split the dataframe into
- **`df`** (*`DataFrame`*) – Any pandas DataFrame
- **`args`** (*`tuple`*) – Positional arguments to pass to the function, keyword arguments can just be passed directly.
- **`profile`** (*`str`*) – A fyrd cluster profile to use
- **`applymap`** (*`bool`*) – Run applymap() instead of apply()
- **`merge_axis`** (*`int`*) – Which axis to merge on, 0 or 1, default is 1 as apply transposes columns
- **`merge_apply`** (*`bool`*) – Apply the function on the merged dataframe also
- **`name`** (*`str`*) – A prefix name for all of the jobs
- **`imports`** (*`list`*) – A list of imports in any format, e.g. ['import numpy', 'scipy', 'from numpy import mean']
- **`direct`** (*`bool`*) – Whether to run the function directly or to return a Job. Default True.
- **`keyword arguments recognized by fyrd will be used for job`** (*`Any`*) –
- **`submission.`** –
- **`keyword arguments will be passed to DataFrame.apply()*`** (*`*Additional`*) –

Returns A recombined DataFrame

Return type DataFrame

`fyrd.helpers.`**`parapply_summary`**(*jobs*, *df*, *func*, *args=()*, *profile=None*, *applymap=False*, *name='parapply'*, *imports=None*, *direct=True*, ***kwds*)

Run parapply for a function with summary stats.

Instead of returning the concatenated result, merge the result using the same function as was used during apply.

This works best for summary functions like *.mean()*, which do a linear operation on a whole dataframe or series.

Parameters

- **`jobs`** (*`int`*) – Number of pieces to split the dataframe into
- **`df`** (*`DataFrame`*) – Any pandas DataFrame
- **`args`** (*`tuple`*) – Positional arguments to pass to the function, keyword arguments can just be passed directly.

- **profile** (*str*) – A fyrd cluster profile to use
- **applymap** (*bool*) – Run applymap() instead of apply()
- **merge_axis** (*int*) – Which axis to merge on, 0 or 1, default is 1 as apply transposes columns
- **merge_apply** (*bool*) – Apply the function on the merged dataframe also
- **name** (*str*) – A prefix name for all of the jobs
- **imports** (*list*) – A list of imports in any format, e.g. ['import numpy', 'scipy', 'from numpy import mean']
- **direct** (*bool*) – Whether to run the function directly or to return a Job. Default True.
- **keyword arguments recognized by fyrd will be used for job** (*Any*) –
- **submission.** –
- **keyword arguments will be passed to DataFrame.apply()*** (**Additional*) –

Returns A recombined DataFrame

Return type DataFrame

fyrd.helpers.**splitrun**(*jobs, infile, inheader, command, args=None, kwargs=None, name=None, qtype=None, profile=None, outfile=None, outheader=False, merge_func=None, direct=True, **kwds*)

Split a file, run command in parallel, return result.

This function will split a file into however many pieces are requested with the jobs argument, and run command on each.

Accepts exactly the same arguments as the Job class, with the exception of the first three and last four arguments, which are:

```
- the number of jobs
- the file to work on
- whether the input file has a header
- an optional output file
- whether the output file has a header
- an optional function to use to merge the resulting list, only used
  if there is no outfile.
- whether to run directly or to return a Job. If direct is True, this
  function will just run and thus block until complete, if direct is
  False, the function will submit as a Job and return that Job.
```

Note: If command is a string, *.format(file={file})* will be called on it, where file is each split file. If command is a function, the there must be an argument in either args or kwargs that contains *{file}*. It will be replaced with the *path to the file*, again by the format command.

If outfile is specified, there must also be an '{outfile}' line in any script or an '{outfile}' argument in either args or kwargs. When this function completes, the file at outfile will contain the concatenated output files of all of the jobs.

If the 'clean_files' and 'clean_outputs' arguments are not passed, we delete all intermediate files and output files by default.

The intermediate files will be stored in the 'scriptpath' directory.

Any header line is kept at the top of the file.

Parameters

- **`jobs`** (*`int`*) – Number of pieces to split the dataframe into
- **`infile`** (*`str`*) – The path to the file to be split.
- **`inheader`** (*`bool`*) – Does the input file have a header?
- **`command`** (*`function/str`*) – The command or function to execute.
- **`args`** (*`tuple/dict`*) – Optional arguments to add to command, particularly useful for functions.
- **`kwargs`** (*`dict`*) – Optional keyword arguments to pass to the command, only used for functions.
- **`name`** (*`str`*) – Optional name of the job. If not defined, guessed. If a job of the same name is already queued, an integer job number (not the queue number) will be added, ie. <name>.1
- **`qtype`** (*`str`*) – Override the default queue type
- **`profile`** (*`str`*) – The name of a profile saved in the conf
- **`outfile`** (*`str`*) – The path to the expected output file.
- **`outheader`** (*`bool`*) – Does the input outfile have a header?
- **`merge_func`** (*`function`*) – An optional function used to merge the output list if there is no outfile.
- **`direct`** (*`bool`*) – Whether to run the function directly or to return a Job. Default True.
- **`other keywords are parsed into cluster keywords by the`** (*`*All`*) –
- **`system. For available keywords see `fyrd.option_help()`*`** (*`options`*) –

Returns

If outfile: the absolute path to that file

If merge_func: the result of merge_func(list), where list is the list of outputs.

Else: a list of results

If direct is False, this function returns a fyrd.job.Job object which will return the results described above on get().

Return type Primary return value varies and is decided in this order

6.7 fyrd.basic

This module holds high level functions to make job submission easy, allowing the user to skip multiple steps and to avoid using the *Job* class directly.

submit(), *make_job()*, and *make_job_file()* all create *Job* objects in the background and allow users to submit jobs. All of these functions accept the exact same arguments as the *Job* class does, and all of them return a *Job* object.

submit_file() is different, it simply submits a pre-formed job file, either one that has been written by this software or by any other method. The function makes no attempt to fix arguments to allow submission on multiple clusters, it just submits the file.

clean() takes a list of job objects and runs the *clean()* method on all of them, *clean_dir()* uses known directory and suffix information to clean out all job files from any directory.

`fyrd.basic.`**`submit`**`()`

Submit a script to the cluster.

Parameters

- **`command`** (*`function/str`*) – The command or function to execute.
- **`args`** (*`tuple/dict`*) – Optional arguments to add to command, particularly useful for functions.
- **`kwargs`** (*`dict`*) – Optional keyword arguments to pass to the command, only used for functions.
- **`name`** (*`str`*) – Optional name of the job. If not defined, guessed. If a job of the same name is already queued, an integer job number (not the queue number) will be added, ie. <name>.1
- **`qtype`** (*`str`*) – Override the default queue type
- **`profile`** (*`str`*) – The name of a profile saved in the conf
- **`other keywords are parsed into cluster keywords by the`** (*`*All`*) –
- **`system. For available keywords see `fyrd.option_help()`*`** (*`options`*) –

Returns Job object

`fyrd.basic.`**`make_job`**`()`

Make a job file compatible with the chosen cluster.

If mode is local, this is just a simple shell script.

Parameters

- **`command`** (*`function/str`*) – The command or function to execute.
- **`args`** (*`tuple/dict`*) – Optional arguments to add to command, particularly useful for functions.
- **`kwargs`** (*`dict`*) – Optional keyword arguments to pass to the command, only used for functions.
- **`name`** (*`str`*) – Optional name of the job. If not defined, guessed. If a job of the same name is already queued, an integer job number (not the queue number) will be added, ie. <name>.1
- **`qtype`** (*`str`*) – Override the default queue type
- **`profile`** (*`str`*) – The name of a profile saved in the conf
- **`other keywords are parsed into cluster keywords by the`** (*`*All`*) –
- **`system. For available keywords see `fyrd.option_help()`*`** (*`options`*) –

Returns Job object

`fyrd.basic.`**`make_job_file`**`()`

Make a job file compatible with the chosen cluster.

Parameters

- **`command`** (*`function/str`*) – The command or function to execute.

- **args** (*tuple/dict*) – Optional arguments to add to command, particularly useful for functions.
- **kwargs** (*dict*) – Optional keyword arguments to pass to the command, only used for functions.
- **name** (*str*) – Optional name of the job. If not defined, guessed. If a job of the same name is already queued, an integer job number (not the queue number) will be added, ie. <name>.1
- **qtype** (*str*) – Override the default queue type
- **profile** (*str*) – The name of a profile saved in the conf
- **other keywords are parsed into cluster keywords by the** (**All*) –
- **system. For available keywords see `fyrd.option_help()`*** (*options*) –

Returns Job object

`fyrd.basic.`**`submit_file`**`()`

Submit a job file to the cluster.

If qtype or queue.MODE is torque, qsub is used; if it is slurm, sbatch is used; if it is local, the file is executed with subprocess.

This function is independent of the Job object and just submits a file.

Parameters

- **dependencies** – A job number or list of job numbers. In slurm: *–dependency=afterok:* is used For torque: *-W depend=afterok:* is used
- **threads** – Total number of threads to use at a time, defaults to all. ONLY USED IN LOCAL MODE

Returns job number for torque or slurm multiprocessing job object for local mode

`fyrd.basic.`**`clean`**`()`

Delete all files in jobs list or single Job object.

`fyrd.basic.`**`clean_dir`**`()`

Delete all files made by this module in directory.

CAUTION: The clean() function will delete EVERY file with

extensions matching those these:: .<suffix>.err .<suffix>.out .<suffix>.out.func.pickle .<suffix>.sbatch & .<suffix>.script for slurm mode .<suffix>.qsub for torque mode .<suffix> for local mode _func.<suffix>.py _func.<suffix>.py.pickle.in _func.<suffix>.py.pickle.out

Parameters

- **directory** (*str*) – The directory to run in, defaults to the current directory.
- **suffix** (*str*) – Override the default suffix.
- **qtype** (*str*) – Only run on files of this qtype
- **confirm** (*bool*) – Ask the user before deleting the files
- **delete_outputs** (*bool*) – Delete all output files too.

Returns A set of deleted files

6.8 fyrd.local

The local queue implementation is based on the multiprocessing library and is not intended to be used directly, it should always be used via the Job class because it is somewhat temperamental. The essential idea behind it is that we can have one JobQueue class that is bound to the parent process, it exclusively manages a single child thread that runs the *job_runner()* function. The two process communicate using a *multiprocessing.Queue* object, and pass *fyrd.local.Job* objects back and forth between them.

The Job objects (different from the Job objects in *job.py*) contain information about the task to run, including the number of cores required. The job runner manages a pool of *multiprocessing.Pool* tasks directly, and keeps the total running cores below the total allowed (default is the system max, can be set with the threads keyword). It backfills smaller jobs and holds on to larger jobs until there is enough space free.

This is close to what torque and slurm do, but vastly more crude. It serves as a stopgap to allow parallel software written for compute clusters to run on a single machine in a similar fashion, without the need for a pipeline alteration. The reason I have reimplemented a process pool is that I need dependency tracking and I need to allow some processes to run on multiple cores (e.g. 6 of the available 24 on the machine).

The *job_runner()* and *Job* objects should never be accessed except by the JobQueue. Only one JobQueue should run at a time (not enforced), and by default it is bound to *fyrd.local.JQUEUE*. That is the interface used by all other parts of this package.

6.8.1 fyrd.local.JobQueue

class `fyrd.local.`**`JobQueue`**(*cores=None*)

Bases: `object`

Monitor and submit multiprocessing.Pool jobs with dependencies.

Spawn a job_runner process to interact with.

`add`(*function*, *args=None*, *kwargs=None*, *dependencies=None*, *cores=1*)

Add function to local job queue.

Parameters

- **`function`** – A function object. To run a command, use the run.cmd function here.
- **`args`** – A tuple of args to submit to the function.
- **`kwargs`** – A dict of keyword arguments to submit to the function.
- **`dependencies`** – A list of job IDs that this job will depend on.
- **`cores`** – The number of threads required by this job.

Returns A job ID

Return type int

`get`(*job*)

Return the output of a single job

`restart`(*force=False*)

Kill the job queue and restart it.

`update`()

Get fresh job info from the runner.

`wait`(*jobs=None*)

Wait for a list of jobs, all jobs are the default.

6.8.2 fyrd.local.job_runner

`fyrd.local.`**`job_runner`**(*jobqueue*, *outputs*, *cores=None*, *jobno=None*)

Run jobs with dependency tracking.

Must be run as a separate multiprocessing.Process to function correctly.

Parameters

- **`jobqueue`** – A multiprocessing.Queue object into which Job objects must be added. The function continually searches this Queue for new jobs. Note, function must be a function call, it cannot be anything else. function is the only required argument, the rest are optional. tuples are required.
- **`outputs`** – A multiprocessing.Queue object that will take outputs. A dictionary of job objects will be output here with the format:: {job_no => Job} **NOTE**: function return must be picklable otherwise this will raise an exception when it is put into the Queue object.
- **`cores`** – Number of cores to use in the multiprocessing pool. Defaults to all.
- **`jobno`** – What number to start counting jobs from, default 1.

6.9 fyrd.run

A library of useful functions used throughout the *fyrd* package.

These include functions to handle data, format outputs, handle file opening, run commands, check file extensions, get user input, and search and format imports.

These functions are not intended to be accessed directly.

`exception` `fyrd.run.`**`CommandError`**

Bases: `exceptions.Exception`

A custom exception.

`class` `fyrd.run.`**`CustomFormatter`**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: `argparse.ArgumentDefaultsHelpFormatter`, `argparse.RawDescriptionHelpFormatter`

Custom argparse formatting.

`fyrd.run.`**`block_read`**(*files*, *size=65536*)

Iterate through a file by blocks.

`fyrd.run.`**`check_pid`**(*pid*)

Check For the existence of a unix pid.

`fyrd.run.`**`cmd`**(*command*, *args=None*, *stdout=None*, *stderr=None*, *tries=1*)

Run command and return status, output, stderr.

Parameters

- **`command`** (`str`) – Path to executable.
- **`args`** (`tuple`) – Tuple of arguments.
- **`stdout`** (`str`) – File or open file like object to write STDOUT to.
- **`stderr`** (`str`) – File or open file like object to write STDERR to.
- **`tries`** (`int`) – Number of times to try to execute. 1+

Returns exit_code, STDOUT, STDERR

Return type tuple

`fyrd.run.`**`count_lines`**(*infile*, *force_blocks=False*)

Return the line count of a file as quickly as possible.

Uses *wc* if avaialable, otherwise does a rapid read.

`fyrd.run.`**`export_globals`**(*function*)

Add a function's globals to the current globals.

`fyrd.run.`**`export_imports`**(*function*, *kwds*)

Get imports from a function and from kwds.

Also sets globals and adds path to module to sys path.

Parameters

- **`function`** (*`callable`*) – A function handle
- **`kwds`** (*`dict`*) – A dictionary of keyword arguments

Returns imports + sys.path.append for module path

Return type list

`fyrd.run.`**`export_run`**(*function*, *args*, *kwargs*)

Execute a function after first exporting all imports.

`fyrd.run.`**`file_type`**(*infile*)

Return file type after stripping gz or bz2.

`fyrd.run.`**`get_all_imports`**(*function*, *kwds*, *prot=False*)

Get all imports from a function and from kwds.

Parameters

- **`function`** (*`callable`*) – A function handle
- **`kwds`** (*`dict`*) – A dictionary of keyword arguments
- **`prot`** (*`bool`*) – Wrap all import in try statement

Returns Imports

Return type list

`fyrd.run.`**`get_function_path`**(*function*)

Return path to module defining a function if it exists.

`fyrd.run.`**`get_imports`**(*function*, *mode='string'*)

Build a list of potentially useful imports from a function handle.

Gets:

- All modules from globals()
- All modules from the function's globals()
- All functions from the function's globals()

Modes: string: Return a list of strings formatted as unprotected import calls prot: Similar to string, but with try..except blocks list: Return two lists: (import name, module name) for modules

and (import name, function name, module name) for functions

Parameters

- **`function`** (*`callable`*) – A function handle
- **`mode`** (*`str`*) – A string corresponding to one of the above modes

Returns str or list

`fyrd.run.`**`get_input`**(*message*, *valid_answers=None*, *default=None*)

Get input from the command line and check answers.

Allows input to work with python 2/3

Parameters

- **`message`** (*`str`*) – A message to print, an additional space will be added.
- **`valid_answers`** (*`list`*) – A list of answers to accept, if None, ignored. Case insensitive. There is one special option here: 'yesno', this allows all case insensitive variations of y/n/yes/no.
- **`default`** (*`str`*) – The default answer.

Returns The response

Return type str

`fyrd.run.`**`get_yesno`**(*message*, *default=None*)

Get yes/no answer from user.

Parameters

- **`message`** (*`str`*) – A message to print, an additional space will be added.
- **`default`** (*`str`*) – One of {'y', 'n'}, the default if the user gives no answer. If None, answer forced.

Returns True on yes, False on no

Return type bool

`fyrd.run.`**`import_function`**(*function*, *mode='string'*)

Return an import string for the function.

Attempts to resolve the parent module also, if the parent module is a file, ie it isn't __main__, the import string will include a call to sys.path.append to ensure the module is importable.

If this function isn't defined by a module, returns an empty string.

Parameters **`mode`** (*`str`*) – string/list, return as a unified string or a list.

`fyrd.run.`**`indent`**(*string*, *prefix=' '*)

Replicate python3's textwrap.indent for python2.

Parameters

- **`string`** (*`str`*) – Any string.
- **`prefix`** (*`str`*) – What to indent with.

Returns Indented string

Return type str

`fyrd.run.`**`is_exc`**(*x*)

Check if x is the output of sys.exc_info().

Returns True if matched the output of sys.exc_info().

Return type bool

`fyrd.run.`**`is_exe`**(*fpath*)
Return True is fpath is executable.

`fyrd.run.`**`is_file_type`**(*infile*, *types*)
Return True if infile is one of types.

Parameters

- **`infile`** – Any file name
- **`types`** – String or list/tuple of strings (e.g [‘bed’, ‘gtf’])

Returns True or False

`fyrd.run.`**`listify`**(*iterable*)
Try to force any iterable into a list sensibly.

`fyrd.run.`**`merge_lists`**(*lists*)
Turn a list of lists into a single list.

`fyrd.run.`**`normalize_imports`**(*imports*, *prot=True*)
Take a heterogenous list of imports and normalize it.

Parameters

- **`imports`** (*`list`*) – A list of strings, formatted differently.
- **`prot`** (*`bool`*) – Protect imports with try..except blocks

Returns A list of strings that can be used for imports

Return type list

`fyrd.run.`**`open_zipped`**(*infile*, *mode='r'*)
Open a regular, gzipped, or bz2 file.

If infile is a file handle or text device, it is returned without changes.

Returns text mode file handle.

`fyrd.run.`**`opt_split`**(*opt*, *split_on*)
Split options by chars in split_on, merge all into single list.

Parameters

- **`opt`** (*`list`*) – A list of strings, can be a single string.
- **`split_on`** (*`list`*) – A list of characters to use to split the options.

Returns

A single merged list of split options, uniqueness guaranteed, order not.

Return type list

`fyrd.run.`**`replace_argument`**(*args*, *find_string*, *replace_string*, *error=True*)
Replace find_string with replace string in a tuple or dict.

If dict, the values are replaced, not the keys.

Note: args can also be a list, in which case the first item is assumed to be a tuple, and the second a dictionary

Parameters

- **`args`** (*`list/tuple/dict`*) – Tuple or dict of args
- **`find_string`** (*`str`*) – A string to search for

- **`replace_string`** (*`str`*) – A string to replace with
- **`error`** (*`bool`*) – Raise ValueError if replacement fails

Returns The same object as was passed, with alterations made.

`fyrd.run.`**`split_file`**(*infile*, *parts*, *outpath=''*, *keep_header=False*)
Split a file in parts and return a list of paths.

NOTE: Linux specific (uses wc).

Note: If has_header is True, the top line is stripped off the infile prior to splitting and assumed to be the header.

Parameters

- **`outpath`** – The directory to save the split files.
- **`has_header`** – Add the header line to the top of every file.

Returns Paths to split files.

Return type list

`fyrd.run.`**`syspath_fmt`**(*syspaths*)
Take a list of paths and return a sys of sys.path.append strings.

`fyrd.run.`**`update_syspaths`**(*function*, *kwds=None*)
Add function path to 'syspaths' in kwds.

`fyrd.run.`**`which`**(*program*)
Replicate the UNIX which command.

Taken verbatim from: stackoverflow.com/questions/377017/test-if-executable-exists-in-python

Parameters **`program`** – Name of executable to test.

Returns Path to the program or None on failu_re.

`fyrd.run.`**`write_iterable`**(*iterable*, *outfile*)
Write all elements of iterable to outfile.

6.10 fyrd.logme

This is a package I wrote myself and keep using because I like it. It provides syslog style leveled logging (e.g. 'debug'->'info'->'warn'->'error'->'critical') and it implements colors and timestamped messages.

The minimum print level can be set module wide at runtime by changing *cluster.logme.MIN_LEVEL*.

`fyrd.logme.`**`log`**(*message*, *level='info'*, *logfile=None*, *also_write=None*, *min_level=None*, *kind=None*)
Print a string to logfile.

Parameters

- **`message`** – The message to print.
- **`logfile`** – Optional file to log to, defaults to STDERR. Can provide a logging object
- **`level`** – 'debug'|'info'|'warn'|'error'|'normal' Will only print if level > MIN_LEVEL

'verbose':	'<timestamp> VERBOSE –> '
'debug':	'<timestamp> DEBUG –> '
'info':	'<timestamp> INFO –> '
'warn':	'<timestamp> WARNING –> '
'error':	'<timestamp> ERROR –> '
'critical':	'<timestamp> CRITICAL –> '

- **also_write** – 'stdout': print to STDOUT also. 'stderr': print to STDERR also. These only have an effect if the output is not already set to the same device.
- **min_level** – Retained for backwards compatibility, min_level should be set using the logme.MIN_LEVEL constant.
- **kind** – synonym for level, kept to retain backwards compatibility

Logging with timestamps and optional log files.

Print a timestamped message to a logfile, STDERR, or STDOUT.

If STDERR or STDOUT are used, colored flags are added. Colored flags are INFO, WARNINING, ERROR, or CRITICAL.

It is possible to write to both logfile and STDOUT/STDERR using the also_write argument.

If level is 'error' or 'critical', error is written to STDERR unless also_write == -1

MIN_LEVEL can also be provided, logs will only print if vlevel > MIN_LEVEL. Level order: critical>error>warn>info>debug>verbose

Usage:

```
import logme as lm
lm.log("Screw up!", <outfile>,
    level='debug'|'info'|'warn'|'error'|'normal',
    also_write='stderr'|'stdout')
```

Example:

```
lm.log('Hi')
Prints: 20160223 11:46:24.969 | INFO --> Hi
lm.log('Hi', level='debug')
Prints nothing
lm.MIN_LEVEL = 'debug'
lm.log('Hi', level='debug')
Prints: 20160223 11:46:24.969 | DEBUG --> Hi
```

Note: Uses terminal colors and STDERR, not compatible with non-unix systems

`fyrd.logme.`**`log`**(*message*, *level='info'*, *logfile=None*, *also_write=None*, *min_level=None*, *kind=None*)

Print a string to logfile.

Parameters

- **message** – The message to print.
- **logfile** – Optional file to log to, defaults to STDERR. Can provide a logging object
- **level** – 'debug'|'info'|'warn'|'error'|'normal' Will only print if level > MIN_LEVEL

'verbose':	'<timestamp> VERBOSE –> '
'debug':	'<timestamp> DEBUG –> '
'info':	'<timestamp> INFO –> '
'warn':	'<timestamp> WARNING –> '
'error':	'<timestamp> ERROR –> '
'critical':	'<timestamp> CRITICAL –> '

- **`also_write`** – 'stdout': print to STDOUT also. 'stderr': print to STDERR also. These only have an effect if the output is not already set to the same device.
- **`min_level`** – Retained for backwards compatibility, min_level should be set using the logme.MIN_LEVEL constant.
- **`kind`** – synonym for level, kept to retain backwards compatibility

Python Module Index

f

fyrd.logme, 42
fyrd.run, 37

Index

A

active_job_count (Queue attribute), 19
add() (fyrd.local.JobQueue method), 36

B

block_read() (in module fyrd.run), 37

C

can_submit (Queue attribute), 20
check_arguments() (in module fyrd.options), 27
check_pid() (in module fyrd.run), 37
check_queue() (in module fyrd.queue), 21
clean() (fyrd.job.Job method), 23
clean() (fyrd.submission_scripts.Function method), 26
clean() (fyrd.submission_scripts.Script method), 26
clean() (in module fyrd.basic), 35
clean() (Job method), 22
clean_dir() (in module fyrd.basic), 35
cmd() (in module fyrd.run), 37
CommandError, 37
count_lines() (in module fyrd.run), 38
create_config() (in module fyrd.conf), 29
create_config_interactive() (in module fyrd.conf), 29
CustomFormatter (class in fyrd.run), 37

D

delete() (in module fyrd.conf), 29
done (Job attribute), 22

E

end (Job attribute), 22
exists (fyrd.submission_scripts.Script attribute), 26
exitcode (Job attribute), 22
export_globals() (in module fyrd.run), 38
export_imports() (in module fyrd.run), 38
export_run() (in module fyrd.run), 38

F

fetch_outputs() (fyrd.job.Job method), 25
file_type() (in module fyrd.run), 38
files (Job attribute), 22
Function (class in fyrd.submission_scripts), 26
fyrd.logme (module), 42
fyrd.run (module), 37

G

get() (fyrd.job.Job method), 24
get() (fyrd.local.JobQueue method), 36
get() (Job method), 22
get_all_imports() (in module fyrd.run), 38
get_cluster_environment() (in module fyrd.queue), 21
get_exitcode() (fyrd.job.Job method), 25
get_function_path() (in module fyrd.run), 38
get_imports() (in module fyrd.run), 38
get_input() (in module fyrd.run), 39
get_jobs() (fyrd.queue.Queue method), 20
get_option() (in module fyrd.conf), 28
get_output() (fyrd.job.Job method), 24
get_profile() (in module fyrd.conf), 30
get_stderr() (fyrd.job.Job method), 24
get_stdout() (fyrd.job.Job method), 24
get_times() (fyrd.job.Job method), 25
get_yesno() (in module fyrd.run), 39

I

import_function() (in module fyrd.run), 39
indent() (in module fyrd.run), 39
is_exc() (in module fyrd.run), 39
is_exe() (in module fyrd.run), 40
is_file_type() (in module fyrd.run), 40

J

Job (class in fyrd), 22
job_runner() (in module fyrd.local), 37
job_states (Queue attribute), 19
JobQueue (class in fyrd.local), 36
jobs (Queue attribute), 19

L

listify() (in module fyrd.run), 40
load_config() (in module fyrd.conf), 29
log() (in module fyrd.logme), 41, 42

M

make_job() (in module fyrd.basic), 34
make_job_file() (in module fyrd.basic), 34
max_jobs (Queue attribute), 19
merge_lists() (in module fyrd.run), 40

N

normalize_imports() (in module fyrd.run), 40

O

open_zipped() (in module fyrd.run), 40
opt_split() (in module fyrd.run), 40
option_help() (in module fyrd.options), 27
option_to_string() (in module fyrd.options), 28
options_to_string() (in module fyrd.options), 27
out (Job attribute), 22

P

parapply() (in module fyrd.helpers), 30
parapply_summary() (in module fyrd.helpers), 31
Profile (class in fyrd.conf), 30

Q

Queue (class in fyrd.queue), 19
queue_parser() (in module fyrd.queue), 20
QueueError, 20

R

replace_argument() (in module fyrd.run), 40
restart() (fyrd.local.JobQueue method), 36
runtime (Job attribute), 22

S

sanitize_arguments() (in module fyrd.options), 27
Script (class in fyrd.submission_scripts), 25
set_option() (in module fyrd.conf), 28
set_profile() (in module fyrd.conf), 30
slurm_queue_parser() (in module fyrd.queue), 21
split_file() (in module fyrd.run), 41
split_keywords() (in module fyrd.options), 27
splitrun() (in module fyrd.helpers), 32
start (Job attribute), 22
stderr (Job attribute), 22
stdout (Job attribute), 22
submit() (fyrd.job.Job method), 23
submit() (in module fyrd.basic), 34
submit() (Job method), 22
submit_file() (in module fyrd.basic), 35
syspath_fmt() (in module fyrd.run), 41

T

torque_queue_parser() (in module fyrd.queue), 21

U

update() (fyrd.job.Job method), 25
update() (fyrd.local.JobQueue method), 36
update() (fyrd.queue.Queue method), 20
update_queue_info() (fyrd.job.Job method), 25
update_syspaths() (in module fyrd.run), 41

W

wait() (fyrd.job.Job method), 24
wait() (fyrd.local.JobQueue method), 36
wait() (fyrd.queue.Queue method), 20
wait() (Job method), 22
wait_to_submit() (fyrd.queue.Queue method), 20
which() (in module fyrd.run), 41
write() (fyrd.conf.Profile method), 30
write() (fyrd.job.Job method), 23
write() (fyrd.submission_scripts.Function method), 26
write() (fyrd.submission_scripts.Script method), 26
write_config() (in module fyrd.conf), 29
write_iterable() (in module fyrd.run), 41