nestly Documentation

Release 0.6.1

Erick Matsen et al.

Apr 23, 2017

Contents

1 Examples **3**
- 1.1 Comparing two algorithms . . . 3
- 1.2 Building Nests . . . 7
- 1.3 SCons integration . . . 10

2 nestly Package **13**
- 2.1 `nestly` Package . . . 13
- 2.2 `core` Module . . . 13
- 2.3 `scons` Module . . . 14
- 2.4 Subpackages . . . 16

3 Command line tools **19**
- 3.1 `nestrun` . . . 19
- 3.2 `nestagg` . . . 20

4 SCons integration **23**
- 4.1 Constructing an `SConsWrap` . . . 23
- 4.2 Adding levels . . . 23
- 4.3 Adding targets . . . 24
- 4.4 Adding aggregates . . . 25
- 4.5 Calling commands from SCons . . . 26

5 Project Modules **29**

6 Changes **31**
- 6.1 0.6.1 . . . 31
- 6.2 0.6.0 . . . 31
- 6.3 0.5.0 . . . 31
- 6.4 0.4.0 . . . 31
- 6.5 0.3.0 . . . 32
- 6.6 0.2.0 . . . 32

7 Indices and tables **33**

Python Module Index **35**

`nestly` is a small package designed to ease running software with combinatorial choices of parameters. It can easily do so for "cartesian products" of parameter choices, but can do much more– arbitrary "backwards-looking" dependencies can be used.

To find out more, check out the the *Examples*.

Contents:

CHAPTER 1

Examples

Comparing two algorithms

This is a realistic example of using `nestly` to examine the performance of two algorithms. Source code to run it is available in `examples/adcl/`.

We will use the `min_adcl_tree` subcommand of the `rppr` tool from the `pplacer` suite, available from http://matsen.fhcrc.org/pplacer.

This tool chooses `k` representative leaves from a phylogenetic tree. There are two implementations: the **Full** algorithm solves the problem exactly, while the **PAM** algorithm uses a variation on the partitioning among medoids heuristic to find a solution.

We'd like to compare the two algorithms on a variety of trees, using different values for `k`.

Making the nest

Setting up the comparison is demonstrated in `00make_nest.py`, which builds up combinations of `(algorithm, tree, k)`:

```
1  #!/usr/bin/env python
2
3  # This example compares runtimes of two implementations of
4  # an algorithm to minimize the average distance to the closest leaf
5  # (Matsen et. al., accepted to Systematic Biology).
6  #
7  # To run it, you'll need the `rppr` binary on your path, distributed as part of
8  # the pplacer suite. Source code, or binaries for OS X and 64-bit Linux are
9  # available from http://matsen.fhcrc.org/pplacer/.
10 #
11 # The `rppr min_adcl_tree` subcommand takes a tree, an algorithm name, and
12 # the number of leaves to keep.
13 #
14 # We wish to explore the runtime, over each tree, for various leaf counts.
```

```python

import glob
from os.path import abspath

from nestly import Nest, stripext

# The `trees` directory contains 5 trees, each with 1000 leaves.
# We want to run each algorithm on all of them.
trees = [abspath(f) for f in glob.glob('trees/*.tre')]

n = Nest()

# We'll try both algorithms
n.add('algorithm', ['full', 'pam'])
# For every tree
n.add('tree', trees, label_func=stripext)

# Store the number of leaves - always 1000 here
n.add('n_leaves', [1000], create_dir=False)

# Now we vary the number of leaves to keep (k)
# Sample between 1 and the total number of leaves.
def k(c):
    n_leaves = c['n_leaves']
    return range(1, n_leaves, n_leaves // 10)

# Add `k` to the nest.
# This will call k with each combination of (algorithm, tree, n_leaves).
# Each value returned will be used as a possible value for `k`
n.add('k', k)

# Build the nest:
n.build('runs')
```

Running that:

```
$ ./00make_nest.py
```

Creates a new directory, `runs`.

Within this directory are subdirectories for each algorithm:

```
runs/full
runs/pam
```

Each of these contains a directory for each tree used:

```
$ ls runs/pam
random001  random002  random003  random004  random005
```

Within each of *these* subdirectories are directories for each choice of `k`.

```
$ ls runs/pam/random001
1  101  201  301  401  501  601  701  801  901
```

These directories are leaves. There is a JSON file in each, containing the choices made. For example, `runs/full/random003/401/control.json` contains:

```
{
  "algorithm": "full",
  "tree": "/home/cmccoy/development/nestly/examples/adcl/trees/random003.tre",
  "n_leaves": 1000,
  "k": 401
}
```

Running the algorithm

The `nestrun` command-line tool allows you to run a command for each combination of parameters in a nest. It allows you to substitute parameters chosen by surrounding them in curly brackets, e.g. `{algorithm}`.

To see how long, and how much memory each run uses, we'll use the short shell script `time_rppr.sh`:

```
1 #!/bin/sh
2
3 export TIME='elapsed,maxmem,exitstatus\n%e,%M,%x'
4
5 /usr/bin/time -o time.csv \
6   rppr min_adcl_tree --algorithm {algorithm} --leaves {k} {tree}
```

Note the placeholders for the parameters to be provided at runtime: `k`, `tree`, and `algorithm`.

Running a script like `time_rppr.sh` on every experiment within a nest in parallel is facilitated by the `nestrun` script distributed with `nestly`:

```
$ nestrun -j 4 --template-file time_rppr.sh -d runs
```

(this will take awhile)

This command runs the shell script `time_rppr.sh` for each parameter choice, substituting the appropriate parameters. The `-j 4` flag indicates that 4 processors should be used.

Aggregating results

Now we have a little CSV file in each leaf directory, containing the running time:

```
|----------+--------+-------------|
|  elapsed | maxmem | exitstatus  |
|----------+--------+-------------|
|  17.78   | 471648 | 0           |
|----------+--------+-------------|
```

To analyze these en-masse, we need to combine them and add information about the parameters used to generate them. The `nestagg` script does just this.

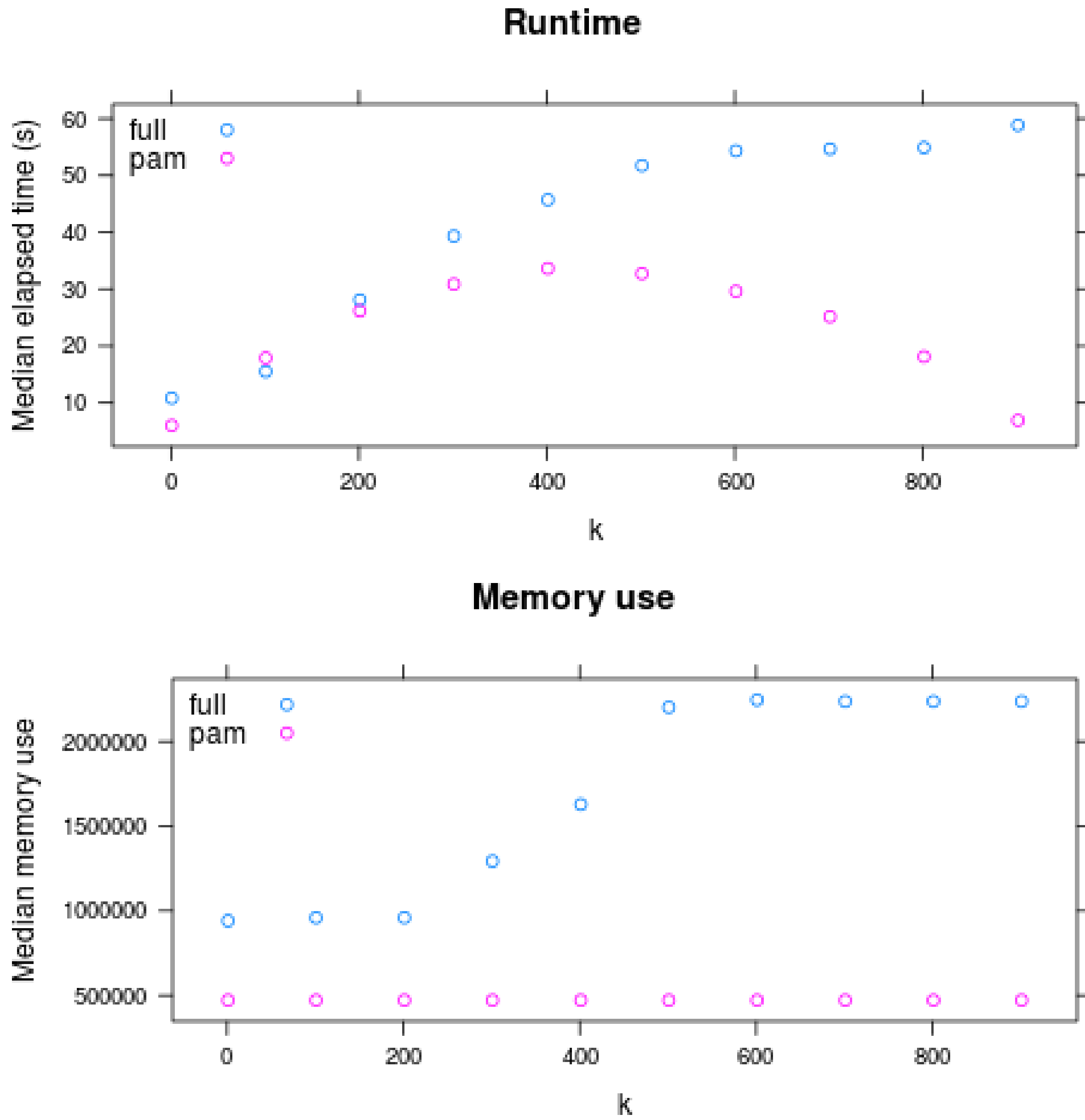
```
$ nestagg delim -d runs -o results.csv time.csv -k algorithm,k,tree
```

Where `-d runs` indicates the directory containing program runs; `-o results.csv` specifies where to write the output; `time.csv` gives the name of the file in each leaf directory, and `-k algorithm,k,tree` lists the parameters to add to each row of the CSV files.

Looking at `results.csv`:

```
|----------+---------+------------+-----------+------------------------------------
↪+------|
|  elapsed | maxmem  | exitstatus | algorithm | tree                               ␣
↪| k     |
|----------+---------+------------+-----------+------------------------------------
↪+------|
|  17.04   | 941328  | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 1     |
|  20.86   | 944336  | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 101   |
|  31.75   | 944320  | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 201   |
|  39.34   | 980048  | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 301   |
|  37.84   | 1118960 | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 401   |
|  42.15   | 1382000 | 0          | full      | .../examples/adcl/trees/random001.tre␣
↪| 501   |
etc
```

Now we have something we can look at!



So: PAM is faster for large `k`, and always has lower maximum memory use.

(generated by `examples/adcl/03analyze.R`)

Building Nests

Basic Nest

From `examples/basic_nest/make_nest.py`, this is a simple, combinatorial example.

```
1 #!/usr/bin/env python
2 
3 import glob
```

```
import math
import os
import os.path
from nestly import Nest

wd = os.getcwd()
input_dir = os.path.join(wd, 'inputs')

nest = Nest()

# Simplest case: Levels are added with a name and an iterable
nest.add('strategy', ('exhaustive', 'approximate'))

# Sometimes it's useful to add multiple keys to the nest in one operation, e.g.
# for grouping related data.
# This can be done by passing an iterable of dictionaries to the `Nest.add` call,
# each containing at least the named key, along with the `update=True` flag.
#
# Here, 'run_count' is the named key, and will be used to create a directory in the␣
↪nest,
# and the value of 'power' will be added to each control dictionary as well.
nest.add('run_count', [{'run_count': 10**i, 'power': i}
                       for i in range(3)], update=True)

# label_func can be used to generate a meaningful name. Here, it strips the all
# but the file name from the file path
nest.add('input_file', glob.glob(os.path.join(input_dir, 'file*')),
        label_func=os.path.basename)

# Items can be added that don't generate directories
nest.add('base_dir', [os.getcwd()], create_dir=False)

# Any function taking one argument (control dictionary) and returning an
# iterable may also be used.
# This one just takes the logarithm of 'run_count'.
# Since the function only returns a single result, we don't create a new directory.
def log_run_count(c):
    run_count = c['run_count']
    return [math.log(run_count, 10)]
nest.add('run_count_log', log_run_count, create_dir=False)

nest.build('runs')
```

This example is then run with the `../examples/basic_nest/run_example.sh` script.

```
#!/bin/sh

set -e
set -u
set -x

# Build a nested directory structure
./make_nest.py

# Let's look at a sample control file:
cat runs/approximate/1/file1/control.json

# Run `echo.sh` using every control.json under the `runs` directory, 2
```

```
# processes at a time
nestrun --processes 2 --template-file echo.sh -d runs

# Merge the CSV files named '{strategy}.csv' (where strategy value is taken
# from the control file)
nestagg delim '{strategy}.csv' -d runs -o aggregated.csv
```

`echo.sh` is just the simple script that runs `nestrun` and aggregates the results into an `aggregate.csv` file:

```
#!/bin/sh
#
# Echo the value of two fake output variables: var1, which is always 13, and
# var2, which is 10 times the run_count.

echo "var1,var2
13,{run_count}0" > "{strategy}.csv"
```

Meal

This is a bit more complicated, with lookups on previous values of the control dictionary:

```
#!/usr/bin/env python

import glob
import os
import os.path

from nestly import Nest, stripext

wd = os.getcwd()
startersdir = os.path.join(wd, "starters")
winedir = os.path.join(wd, "wine")
mainsdir = os.path.join(wd, "mains")

nest = Nest()

bn = os.path.basename

# Start by mirroring the two directory levels in startersdir, and name those
# directories "ethnicity" and "dietary".
nest.add('ethnicity', glob.glob(os.path.join(startersdir, '*')),
    label_func=bn)
# In the `dietary` key, the anonymous function `lambda ...` chooses as values
# names of directories the current `ethnicity` directory
nest.add('dietary', lambda c: glob.glob(os.path.join(c['ethnicity'], '*')),
    label_func=bn)

## Now get all of the starters.
nest.add('starter', lambda c: glob.glob(os.path.join(c['dietary'], '*')),
    label_func=stripext)
## Then get the corresponding mains.
nest.add('main', lambda c: [os.path.join(mainsdir, bn(c['ethnicity']) + "_stirfry.txt
↪")],
    label_func=stripext)

## Take only the tasty wines.
```

```
nest.add('wine', glob.glob(os.path.join(winedir, '*.tasty')),
    label_func=stripext)
## The wineglasses should be chosen by the wine choice, but we don't want to
## make a directory for those.
nest.add('wineglass', lambda c: [stripext(c['wine']) + ' wine glasses'],
        create_dir=False)

nest.build('runs')
```

SCons integration

This `SConstruct` file is an example of using nestly with the SCons build system:

```
# -*- python -*-
#
# This example takes every file in the inputs directory and performs the
# following operations:
# * cuts out a column range from every line in the file; either 1-5 or 3-40
# * optionally filters out every line that has an "o" or "O"
# * runs wc on every such file
# * aggregate these together using the prep_tab.sh script
#
# Assuming that SCons is installed, you should be able to run this example by
# typing `scons` in this directory. That should build a series of things in the
# `build` directory. Because this is a build system, deleting a file or directory
# in the build directory and then running scons will simply rerun the needed parts.

from os.path import join
import os

from nestly.scons import SConsWrap
from nestly import Nest

env = Environment()

# Passing an argument to `alias_environment` allows building targets based on nest
# key.
# For example, the `counts` files described below can be built by invoking
# `scons counts`
nest = SConsWrap(Nest(), 'build', alias_environment=env)


# Add our aggregate targets, initializing collections that will get populated
# downstream. At the end of the pipeline, we will operate on these collections.
# The `add_argument` takes a key which will be the key used for accessing the
# collection. The `list` argument specifies that the collection will be a list.
nest.add_aggregate('count_agg', list)
nest.add_aggregate('cut_agg', list)

# Add a nest level with the name 'input_file' that takes the files in the inputs
# directory as its nestable list. Make its label function just the basename.
nest.add('input_file', [join('inputs', f) for f in os.listdir('inputs')],
      label_func=os.path.basename)

# This nest level determines the column range we will cut out of the file.
```

```
43  nest.add('cut_range', ['1-5', '3-40'])
44
45  # This adds a nest item with the name 'cut' and makes an SCons target out of
46  # the result.
47  @nest.add_target()
48  def cut(outdir, c):
49      cut, = Command(join(outdir, 'cut'),
50                     c['input_file'],
51                     'cut -c {0[cut_range]} <$SOURCE >$TARGET'.format(c))
52      # Here we add this cut file to the all_cut aggregator before returning
53      c['cut_agg'].append(cut)
54      return cut
55
56  # This nest level determines whether we remove the lines with o's.
57  nest.add('o_choice', ['remove_o', 'leave_o'])
58
59  @nest.add_target()
60  def o_choice(outdir, c):
61      # If we leave the o lines, then we don't have to do anything.
62      if c['o_choice'] == 'leave_o':
63          return c['cut']
64      # If we want to remove the o lines, then we have to make an SCons Command
65      # that does so with sed.
66      return Command(join(outdir, 'o_removed'),
67                     c['cut'],
68                     'sed "/[oO]/d" <$SOURCE >$TARGET')[0]
69
70  # Add a target for the word counts.
71  @nest.add_target()
72  def counts(outdir, c):
73      counts, = Command(join(outdir, 'counts'),
74                        c['o_choice'],
75                        'wc <$SOURCE >$TARGET')
76      # Add the resulting file to the count_agg collection
77      c['count_agg'].append(counts)
78      return counts
79
80  # Add a control dictionary with chosen values to each leaf directory
81  nest.add_controls(env)
82
83  # Before operating on our aggregate collections, we return back to the original
84  # nest level in which the aggregates were created by using the `pop` function to
85  # remove all of the later nest levels from the nest state, leaving only the
86  # collections.
87  nest.pop('input_file')
88
89  # Now, back at the initial nest level, we can operate on the populated aggregate
90  # collections. First, the counts:
91  @nest.add_target()
92  def all_counts(outdir, c):
93      return Command(join(outdir, 'all_counts.tab'),
94                     c['count_agg'],
95                     './prep_tab.sh $SOURCES | column -t >$TARGET')
96
97  # Then the cuts:
98  @nest.add_target()
99  def all_cut(outdir, c):
100     return Command(join(outdir, 'all_cut.txt'),
```

```
101                    c['cut_agg'],
102                    'cat $SOURCES >$TARGET')
```

CHAPTER 2

nestly Package

nestly Package

nestly is a collection of functions designed to make running software with combinatorial choices of parameters easier.

core Module

Core functions for building nests.

class nestly.core.**Nest**(*control_name='control.json'*, *indent=2*, *fail_on_clash=False*, *warn_on_clash=True*, *base_dict=None*, *include_outdir=True*)

Bases: object

Nests are used to build nested parameter selections, culminating in a directory structure representing choices made, and a JSON dictionary with all selections.

Build parameter combinations with Nest.add(), then create a nested directory structure with Nest.build().

Parameters

- **control_name** – Name JSON file to be created in each leaf
- **indent** – Indentation level in json file
- **fail_on_clash** – Error if a nest level attempts to overwrite a previous value
- **warn_on_clash** – Print a warning if a nest level attempts ot overwrite a previous value
- **base_dict** – Base dictionary to start all control dictionaries from (default: {})
- **include_outdir** – If true, include an OUTDIR key in every control indicating the directory this control would be written to.

add(*name*, *nestable*, *create_dir=True*, *update=False*, *label_func=<type 'str'>*, *template_subs=False*)

Add a level to the nest

Parameters

- **name** (*string*) – Name of the level. Forms the key in the output dictionary.
- **nestable** – Either an iterable object containing values, _or_ a function which takes a single argument (the control dictionary) and returns an iterable object containing values
- **create_dir** (*boolean*) – Should a directory level be created for this nestable?
- **update** (*boolean*) – Should the control dictionary be updated with the results of each value returned by the nestable? Only valid for dictionary results; useful for updating multiple values. At a minimum, a key-value pair corresponding to `name` must be returned.
- **label_func** – Function to be called to convert each value to a directory label.
- **template_subs** (*boolean*) – Should the strings in / returned by nestable be treated as templates? If true, str.format is called with the current values of the control dictionary.

build(*root='runs'*)

Build a nested directory structure, starting in `root`

Parameters **root** – Root directory for structure

iter(*root=None*)

Create an iterator of (directory, control_dict) tuples for all valid parameter choices in this `Nest`.

Parameters **root** – Root directory

Return type Generator of `(directory, control_dictionary)` tuples.

nestly.core.**control_iter**(*base_dir*, *control_name='control.json'*)

Generate the names of all control files under base_dir

nestly.core.**nest_map**(*control_iter*, *map_fn*)

Apply `map_fn` to the directories defined by `control_iter`

For each control file in control_iter, map_fn is called with the directory and control file contents as arguments.

Example:

```
>>> list(nest_map(['run1/control.json', 'run2/control.json'],
...               lambda d, c: c['run_id']))
[1, 2]
```

Parameters

- **control_iter** – Iterable of paths to JSON control files
- **map_fn** (*function*) – Function to run for each control file. It should accept two arguments: the directory of the control file and the json-decoded contents of the control file.

Returns A generator of the results of applying `map_fn` to elements in `control_iter`

nestly.core.**stripext**(*path*)

Return the basename, minus extension, of a path.

Parameters **path** (*string*) – Path to file

`scons` Module

SCons integration for nestly.

class `nestly.scons.`**`SConsEncoder`**(*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, encoding='utf-8', default=None*)

Bases: `json.encoder.JSONEncoder`

JSON Encoder which handles SCons objects.

`default`(*obj*)

class `nestly.scons.`**`SConsWrap`**(*nest, dest_dir='.', alias_environment=None*)

Bases: `object`

A Nest wrapper to add SCons integration.

This class wraps a *`Nest`* in order to provide methods which are useful for using nestly with SCons.

A Nest passed to SConsWrap must have been created with `include_outdir=True`, which is the default.

Parameters

- **`nest`** – A *`Nest`* object to wrap
- **`dest_dir`** – The base directory for all output directories.
- **`alias_environment`** – An optional SCons `Environment` object. If present, targets added via *`SConsWrap.add_target()`* will include an alias using the nest key.

`add`(*name, nestable, **kw*)

Adds a level to the nesting and creates a checkpoint that can be reverted to later for aggregation by calling *`SConsWrap.pop()`*.

Parameters

- **`name`** – Identifier for the nest level
- **`nestable`** – A nestable object - see *`Nest.add()`*.
- **`kw`** – Additional parameters to pass to *`Nest.add()`*.

`add_aggregate`(*name, data_fac*)

Add an aggregate target to this nest.

Since nests added after the aggregate can access the construct returned by the factory function value, it can be mutated to provide additional values for use when the decorated function is called.

To do something with the aggregates, you must *`SConsWrap.pop()`* nest levels created between addition of the aggregate and then can add any normal targets you would like which take advantage of the targets added to the data structure.

Parameters

- **`name`** – Name for the target in the nest
- **`data_fac`** – a nullary factory function which will be called immediately for each of the current control dictionaries and stored in each dictionary with the given name as in *`SConsWrap.add_target()`*.

`add_controls`(*env, target_name='control', file_name='control.json', encoder_cls=<class 'nestly.scons.SConsEncoder'>*)

Adds a target to build a control file at each of the current leaves.

Parameters

- **`env`** – SCons Environment object
- **`target_name`** – Name for target in nest

- **file_name** – Name for output file.

add_nest(*name=None*, ***kw*)
A simple decorator which wraps `nestly.core.Nest.add()`.

add_target(*name=None*)
Add an SCons target to this nest.

The function decorated will be immediately called with each of the output directories and current control dictionaries. Each result will be added to the respective control dictionary for later nests to access.

Parameters **name** – Name for the target in the name (default: function name).

add_target_with_env(*environment*, *name=None*)
Add an SCons target to this nest, with an SCons Environment

The function decorated will be immediately called with three arguments:

- `environment`: A clone of the SCons environment, with variables populated for all values in the control dictionary, plus a variable `OUTDIR`.
- `outdir`: The output directory
- `control`: The control dictionary

Each result will be added to the respective control dictionary for later nests to access.

Differs from `SConsWrap.add_target()` only by the addition of the `Environment` clone.

pop(*name=None*)
Reverts to the nest stage just before the corresponding call of `SConsWrap.add_aggregate()`. However, any aggregate collections which have been worked on will still be accessible, and can be called operated on together after calling this method. If no name is passed, will revert to the last nest level.

Parameters **name** – Name of the nest level to pop.

`nestly.scons.`**name_targets**(*func*)
Wrap a function such that returning `'a', 'b', 'c', [1, 2, 3]` transforms the value into `dict(a=1, b=2, c=3)`.

This is useful in the case where the last parameter is an SCons command.

Subpackages

scripts Package

nestrun Module

nestrun.py - run commands based on control dictionaries.

class `nestly.scripts.nestrun.`**NestlyProcess**(*command*, *working_dir*, *popen*, *log_name='log.txt'*)
Bases: `object`

Metadata about a process run

complete(*return_code*)
Mark the process as complete with provided return_code

log_tail(*nlines=10*)
Return the last `nlines` lines of the log file

running_time

terminate()

nestly.scripts.nestrun.**extant_file**(*x*)
'Type' for argparse - checks that file exists but does not open.

nestly.scripts.nestrun.**invoke**(*max_procs*, *data*, *json_files*)

nestly.scripts.nestrun.**main**()

nestly.scripts.nestrun.**parse_arguments**()
Grab options and json files.

nestly.scripts.nestrun.**sigint_handler**(*nlocal*, *write_this_summary*, *running_procs*, *signum*, *frame*)

nestly.scripts.nestrun.**sigterm_handler**(*nlocal*, *signum*, *frame*)

nestly.scripts.nestrun.**sigusr1_handler**(*running_procs*, *signum*, *frame*)

nestly.scripts.nestrun.**template_subs_file**(*in_file*, *out_fobj*, *d*)
Substitute template arguments in in_file from variables in d, write the result to out_fobj.

nestly.scripts.nestrun.**worker**(*data*, *json_file*)
Handle parameter substitution and execute command as child process.

nestly.scripts.nestrun.**write_summary**(*all_procs*, *summary_file*)
Write a summary of all run processes to summary_file in tab-delimited format.

nestagg Module

Aggregate results of nestly runs.

nestly.scripts.nestagg.**comma_separated_values**(*s*)

nestly.scripts.nestagg.**delim**(*arguments*)
Execute delim action.

Parameters **arguments** – Parsed command line arguments from main()

nestly.scripts.nestagg.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Command-line interface for nestagg

nestly.scripts.nestagg.**warn**(*message*)

CHAPTER 3

Command line tools

nestrun

nestrun takes a command template and a list of control.json files with variables to substitute. Substitution is performed using the Python built-in str.format method. See the Python Formatter documentation for details on syntax, and examples/jsonrun/do_nestrun.sh for an example.

Signals

nestrun also handles some signals by default.

SIGTERM
This tells nestrun to stop spawning jobs. All jobs that were already spawned will continue running.

SIGINT
This tells nestrun to terminate if received twice. On the first SIGTERM, nestrun will emit a warning message; on the second, it will terminate all jobs and then itself.

SIGUSR1
This tells nestrun to immediately write a list of all currently-running processes and their working directories to stderr, then flush stderr.

Help

```
usage: nestrun.py [-h] [-j N] [--template 'template text'] [--stop-on-error]
                  [--template-file FILE] [--save-cmd-file SAVECMD_FILE]
                  [--log-file LOG_FILE | --no-log] [--dry-run]
                  [--summary-file SUMMARY_FILE] [-d DIR]
                  [control_files [control_files ...]]

nestrun - substitute values into a template and run commands in parallel.
```

```
optional arguments:
  -h, --help            show this help message and exit
  -j N, --processes N, --local N
                        Run a maximum of N processes in parallel locally
                        (default: 2)
  --template 'template text'
                        Command-execution template, e.g. bash {infile}. By
                        default, nestrun executes the templatefile.
  --stop-on-error       Terminate remaining processes if any process returns
                        non-zero exit status (default: False)
  --template-file FILE  Command-execution template file path.
  --save-cmd-file SAVECMD_FILE
                        Name of the file that will contain the command that
                        was executed.
  --log-file LOG_FILE   Name of the file that will contain output of the
                        executed command.
  --no-log              Don't create a log file
  --dry-run             Dry run mode, does not execute commands.
  --summary-file SUMMARY_FILE
                        Write a summary of the run to the specified file

Control files:
  control_files         Nestly control dictionaries
  -d DIR, --directory DIR
                        Run on all control files under DIR. May be used in
                        place of specifying control files.
```

nestagg

The nestagg command provides a mechanism for combining results of multiple runs, via a subcommand interface. Currently, the only supported action is merging delimited files from a set of leaves, adding values from the control dictionary on each. This is performed via nestagg delim.

Help

```
usage: nestagg.py delim [-h] [-k KEYS | -x EXCLUDE_KEYS] [-m {fail,warn}]
                        [-d DIR] [-s SEPARATOR] [-t] [-o OUTPUT]
                        file_template [control.json [control.json ...]]

positional arguments:
  file_template         Template for the delimited file to read in each
                        directory [e.g. '{run_id}.csv']
  control.json          Control files

optional arguments:
  -h, --help            show this help message and exit
  -k KEYS, --keys KEYS  Comma separated list of keys from the JSON file to
                        include [default: all keys]
  -x EXCLUDE_KEYS, --exclude-keys EXCLUDE_KEYS
                        Comma separated list of keys from the JSON file not to
                        include [default: None]
  -m {fail,warn}, --missing-action {fail,warn}
                        Action to take when a file is missing [default: fail]
  -d DIR, --directory DIR
```

```
                        Run on all control files under DIR. May be used in
                        place of specifying control files.
  -s SEPARATOR, --separator SEPARATOR
                        Separator [default: ,]
  -t, --tab             Files are tab-separated
  -o OUTPUT, --output OUTPUT
                        Output file [default: stdout]
```

CHAPTER 4

SCons integration

SCons is an excellent build tool (analogous to `make`). The *`nestly.scons`* module is provided to make integrating nestly with SCons easier. *`SConsWrap`* wraps a *`Nest`* object to provide additional methods for adding nests. SCons is complex and is fully documented on their website, so we do not describe it here. However, for the purposes of this document, it suffices to know that dependencies are created when a *target* function is called.

The basic idea is that when writing an SConstruct file (analogous to a Makefile), these *`SConsWrap`* objects extend the usual nestly functionality with build dependencies. Specifically, there are functions that add targets to the nest. When SCons is invoked, these targets are identified as dependencies and the needed code is run.

Typically, you will only need targets within some nest level to refer to things either in the same nest, or in parent nests. However, it is possible to operate on target collections which are not related in this way by using aggregate targets.

Constructing an `SConsWrap`

`SConsWrap` objects wrap and modify a `Nest` object. Each `Nest` object needs to have been created with `include_outdir=True`, which is the default.

Optionally, a destination directory can be given to the `SConsWrap` which will be passed to *`Nest.iter()`*:

```
>>> nest = SConsWrap(Nest(), dest_dir='build')
```

In this example, all the nests created by `nest` will go under the `build` directory. Throughout the rest of this document, `nest` will refer to this same *`SConsWrap`* instance.

Adding levels

Nest levels can still be added to the `nest` object:

```
>>> nest.add('level1', ['spam', 'eggs'])
```

SConsWrap also provides a convenience decorator *SConsWrap.add_nest()* for adding levels which use a function as their nestable. The following examples are exactly equivalent:

```
@nest.add_nest('level2', label_func=str.strip)
def level2(c):
    return ['  __' + c['level1'], c['level1'] + '__  ']

def level2(c):
    return ['  __' + c['level1'], c['level1'] + '__  ']
nest.add('level2', level2, label_func=str.strip)
```

Another advantage to using the decorator is that the name parameter is optional; if it's omitted, the name of the nest is taken from the name of the function. As a result, the following example is also equivalent:

```
@nest.add_nest(label_func=str.strip)
def level2(c):
    return ['  __' + c['level1'], c['level1'] + '__  ']
```

Note: *add_nest()* must always be called before being applied as a decorator. `@nest.add_nest` is not valid; the correct usage is `@nest.add_nest()` if no other parameters are specified.

Adding targets

The fundamental action of SCons integration is in adding a target to a nest. Adding a target is very much like adding a level in that it will add a key to the control dictionary, except that it will not add any branching to a nest. For example, successive calls to *Nest.add()* produces results like the following

```
>>> nest.add('level1', ['A', 'B'])
>>> nest.add('level2', ['C', 'D'])
>>> pprint.pprint([c.items() for outdir, c in nest])
[[('OUTDIR', 'A/C'), ('level1', 'A'), ('level2', 'C')],
 [('OUTDIR', 'A/D'), ('level1', 'A'), ('level2', 'D')],
 [('OUTDIR', 'B/C'), ('level1', 'B'), ('level2', 'C')],
 [('OUTDIR', 'B/D'), ('level1', 'B'), ('level2', 'D')]]
```

A crude illustration of how `level1` and `level2` relate:

```
#               C .---- - -
#    A .----------o level2
#      |        D '---- - -
# o----o level1
#      |        C .---- - -
#    B '----------o level2
#               D '---- - -
```

Calling *add_target()*, however, produces slightly different results:

```
>>> nest.add('level1', ['A', 'B'])
>>> @nest.add_target()
... def target1(outdir, c):
...     return 't-{0[level1]}'.format(c)
...
>>> pprint.pprint([c.items() for outdir, c in nest])
```

```
[[('OUTDIR', 'A'), ('level1', 'A'), ('target1', 't-A')],
 [('OUTDIR', 'B'), ('level1', 'B'), ('target1', 't-B')]]
```

And a similar illustration of how `level1` and `target1` relate:

```
#                 t-A
#    A .----------o------ - -
# o----o level1       target1
#    B '----------o------ - -
#                 t-B
```

add_target() does not increase the total number of control dictionaries from 2; it only updates each existing control dictionary to add the `target1` key. This is effectively the same as calling *add()* (or *add_nest()*) with a function and returning an iterable of one item:

```
>>> nest.add('level1', ['A', 'B'])
>>> @nest.add_nest()
... def target1(c):
...     return ['t-{0[level1]}'.format(c)]
...
>>> pprint.pprint([c.items() for outdir, c in nest])
[[('OUTDIR', 'A/t-A'), ('level1', 'A'), ('target1', 't-A')],
 [('OUTDIR', 'B/t-B'), ('level1', 'B'), ('target1', 't-B')]]
```

Astute readers might have noticed the key difference between the two: functions decorated with *add_target()* have an additional parameter, `outdir`. This allows targets to be built into the correct place in the directory hierarchy.

The other notable difference is that the function decorated by *add_target()* will be called exactly once with each control dictionary. A function added with *add()* may be called more than once with equal control dictionaries.

Like *add_nest()*, *add_target()* must always be called, and optionally takes the name of the target as the first parameter. No other parameters are accepted.

Adding aggregates

As mentioned in the introduction, often you only need targets within a given nest level to depend on things in the same nest level or parental nest levels. To get around this restriction, you can utilize nestly's aggregate functionality.

Adding an aggregate target creates a collection (for each terminal node of the current nest state) which can be updated in downstream nest levels. Once targets have been added to the aggregate collection, you can return to a previous nest level by using the *pop()* method and operate on the populated aggregate collection at that level.

For example, let's say we have two nest levels, `level1` and `level2`, which take the values `[A, B]` and `[C, D]` respectively. If we want to perform an operation for every unique combination of `{level1, level2}`, then aggregate the results grouped by values of `level1`:

```
>>> # Create the first nest level, and add an aggregate named "aggregate1"
>>> nest.add('level1', ['A', 'B'])
>>> nest.add_aggregate('aggregate1', list)
...
>>> # Next, add level2 and a target to level2
>>> nest.add('level2', ['C', 'D'])
>>> @nest.add_target()
... def some_target(outdir, c):
...     target = c['level1'] + c['level2']
...     # here we populate the aggregate
```

```
...     c['aggregate1'].append(target)
...     return target
...
>>> # Now the aggregates have been filled!
>>> # Note that the aggregate collection is shared among all descendents of
>>> # each `level1` value
>>> pprint.pprint([(c['level1'], c['level2'], c['aggregate1']) for outdir, c in nest])
[('A', 'C', ['AC', 'AD']),
 ('A', 'D', ['AC', 'AD']),
 ('B', 'C', ['BC', 'BD']),
 ('B', 'D', ['BC', 'BD'])]
>>>
>>> # However, if we try to build something from the aggregate collection now, we'd␣
↪get 4 copies (one for
>>> # 'A/C', one for 'A/D', etc.).
>>> # To return to the nest state prior to adding `level2`, we pop it from the nest:
>>> nest.pop('level2')
>>> # Now when we access the aggregate collection, there are only two entries, one␣
↪for A and one for B:
>>> pprint.pprint([(c['level1'], c['aggregate1']) for outdir, c in nest])
[('A', ['AC', 'AD']), ('B', ['BC', 'BD'])]
>>>
>>> # we can add targets using the aggregate collection!
>>> @nest.add_target()
... def operate_on_aggregate(outdir, c):
...     print 'agg', c['level1'], c['aggregate1']
...
agg A ['AC', 'AD']
agg B ['BC', 'BD']
```

As you can see above, aggregate targets are added using the `add_aggregate()` method. The first argument to this method is used as a key for accessing the aggregate collection(s) from the control dictionary. The second argument should be a factory function which will be called with no arguments and set as the initial value of the aggregate (typically a collection constructor like *list* or *dict*).

Prior to using the aggregate collection, any branching nest levels added after the aggregate should be removed, using `pop()` to prevent building identical targets. This function, when passed the name of a nest level, returns the `SConsWrap` to the state just before that nest level was created. The only modifications which remain are those on the aggregate collection, which retains any targets added to it within the removed nest levels. Once back at the parental nest level, targets added to the aggregate can be operated on by any further targets added. Note that to pop a level from the nest, one must call `nestly.scons.SConsWrap.add()` rather than `nestly.core.Nest.add()`.

Because the results of operations on aggregates are just regular targets at some ancestral nest level, these targets can be used as the sources to targets further downstream.

Note: nestly's initial SCons aggregation functionality added in version 0.4.0 and described in the nestly manuscript involved registering aggregate functions before adding additional levels to the nest. This interface did not allow the user to utilize aggregate targets as sources of other targets downstream. The original aggregation functionality has since been removed in favor of that described above.

Calling commands from SCons

While the previous example demonstrate how to use the various methods of `SConsWrap`, they did not demonstrate how to actually call commands using SCons. The easiest way is to define the various targets from within

the `SConstruct` file:

```
from nestly.scons import SConsWrap
from nestly import Nest
import os

nest = Nest()
wrap = SConsWrap(nest, 'build')

# Add a nest for each of our input files.
nest.add('input_file', [join('inputs', f) for f in os.listdir('inputs')],
         label_func=os.path.basename)

# Each input will get transformed each of these different ways.
nest.add('transformation', ['log', 'unit', 'asinh'])

@nest.add_target()
def transformed(outdir, c):
    # The template for the command to run.
    action = 'guppy mft --transform {0[transformation]} $SOURCE -o $TARGET'
    # Command will return a tuple of the targets; we want the only item.
    outfile, = Command(
        source=c['input_file'],
        target=os.path.join(outdir, 'transformed.jplace'),
        action=action.format(c))
    return outfile
```

A function *name_targets()* is also provided for more easily naming the targets of an SCons command:

```
@nest.add_target('target1')
@name_targets
def target1(outdir, c):
    return 'outfile1', 'outfile2', Command(
        source=c['input_file'],
        target=[os.path.join(outdir, 'outfile1'),
                os.path.join(outdir, 'outfile2')],
        action="transform $SOURCE $TARGETS")
```

In this case, `target1` will be a dict resembling `{'outfile1': 'build/outdir/outfile1', 'outfile2': 'build/outdir/outfile2'}`.

Note: *name_targets()* does not preserve the name of the decorated function, so the name of the target *must* be provided as a parameter to *add_target()*.

A more involved, runnable example is in the `examples/scons` directory.

CHAPTER 5

Project Modules

CHAPTER 6

Changes

0.6.1

- Fix bug wherein pop does not work on nest levels added with a function (GH-23).

0.6.0

- Add support for automatic alias creation in `SConsWrap` instances (GH-17).

0.5.0

- Add `SConsWrap.add_target_with_env` (GH-14)
- Completely revamped aggregation functionality (GH-15)
- Add `SConsWrap.add_controls` (GH-16)

0.4.0

- Add `SIG{INT,TERM,USR1}` handling to nestrun (GH-9)
- Add SCons integration via `nestly.scons` (GH-12)
- Support for walking a directory in nestagg (GH-13)
- Initial Python 3, PyPy support
- Add an `OUTDIR` key to nest control files
- Additional examples

0.3.0

- Add `nestly.core.stripext`
- New aggregation functionality: `nestagg` subcommand; `nestly.core.nest_map`
- Show tail of log file when `nestrun` fails (GH-10)

0.2.0

- Deprecated `nestly.nestly`
- New object-oriented API in `nestly.core`
- Updated examples

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

nestly.__init__, 13
nestly.core, 13
nestly.scons, 14
nestly.scripts.nestagg, 17
nestly.scripts.nestrun, 16

Index

A

add() (nestly.core.Nest method), 13
add() (nestly.scons.SConsWrap method), 15
add_aggregate() (nestly.scons.SConsWrap method), 15
add_controls() (nestly.scons.SConsWrap method), 15
add_nest() (nestly.scons.SConsWrap method), 16
add_target() (nestly.scons.SConsWrap method), 16
add_target_with_env() (nestly.scons.SConsWrap method), 16

B

build() (nestly.core.Nest method), 14

C

comma_separated_values() (in module nestly.scripts.nestagg), 17
complete() (nestly.scripts.nestrun.NestlyProcess method), 16
control_iter() (in module nestly.core), 14

D

default() (nestly.scons.SConsEncoder method), 15
delim() (in module nestly.scripts.nestagg), 17

E

extant_file() (in module nestly.scripts.nestrun), 17

I

invoke() (in module nestly.scripts.nestrun), 17
iter() (nestly.core.Nest method), 14

L

log_tail() (nestly.scripts.nestrun.NestlyProcess method), 16

M

main() (in module nestly.scripts.nestagg), 17
main() (in module nestly.scripts.nestrun), 17

N

name_targets() (in module nestly.scons), 16
Nest (class in nestly.core), 13
nest_map() (in module nestly.core), 14
nestly.__init__ (module), 13
nestly.core (module), 13
nestly.scons (module), 14
nestly.scripts.nestagg (module), 17
nestly.scripts.nestrun (module), 16
NestlyProcess (class in nestly.scripts.nestrun), 16

P

parse_arguments() (in module nestly.scripts.nestrun), 17
pop() (nestly.scons.SConsWrap method), 16

R

running_time (nestly.scripts.nestrun.NestlyProcess attribute), 16

S

SConsEncoder (class in nestly.scons), 14
SConsWrap (class in nestly.scons), 15
sigint_handler() (in module nestly.scripts.nestrun), 17
sigterm_handler() (in module nestly.scripts.nestrun), 17
sigusr1_handler() (in module nestly.scripts.nestrun), 17
stripext() (in module nestly.core), 14

T

template_subs_file() (in module nestly.scripts.nestrun), 17
terminate() (nestly.scripts.nestrun.NestlyProcess method), 17

W

warn() (in module nestly.scripts.nestagg), 17
worker() (in module nestly.scripts.nestrun), 17
write_summary() (in module nestly.scripts.nestrun), 17