

Ducttape by Example

Jonathan Clark

Contents

1	Introduction	1
2	Basics	2
2.1	Basics: Running a single command	2
2.2	Basics: Writing output files	2
2.3	Basics: Reading input files	3
2.4	Basics: Running tasks with dependencies	3
2.5	Basics: Using parameters	3
2.6	Basics: External config files	4
2.7	Basics: Global variables	4
2.8	Basics: Shorthand variable references	5
3	Packages	5
3.1	Packages: Understanding the git versioner	5
3.2	Adapting for Your Environment	6
4	HyperWorkflows	6
4.1	HyperWorkflows: What are HyperWorkflows?	6
4.2	HyperWorkflows: The default one off plan	7
4.3	HyperWorkflows: Sequence Branch Points	8
4.4	HyperWorkflows: Custom realization plans	8
4.5	HyperWorkflows: Branch Grating	9
4.6	HyperWorkflows: Nested Branch Points	9
4.7	Running Multiple Instances (Processes) of Ducttape	10
5	Submitters	10
5.1	Submitters: Shell	10
5.2	Submitters: Sun Grid Engine (Simple)	10
5.3	Appendix A: Directory Structure	11
5.4	Destination Directory	11
5.5	Packages	11
5.6	The Attic	11
5.7	"Flat" Structure	12
5.8	"Hyper" Structure	12
5.9	Config Directories	12
5.10	Transitioning from "Flat" to "Hyper"	12
5.11	What directives are available in Ducttape?	12
5.12	Why are tasks more like Make targets than functions?	12
5.13	Job control	13

1 Introduction

All examples in this tutorial are runnable and can be found in in the directory tutorial/

2 Basics

2.1 Basics: Running a single command

Syntax:

This is just a single command that writes output to stdout and stderr

Lines that begin with `#` are comments

Comment blocks preceding task declarations are associated with that task

Execution:

After ducttape has been added to your PATH, this workflow can be executed with:

```
$ ducttape basic.tape
```

Artifacts from this workflow will be in the current directory of basic.tape (`./`)

This task will run in the working directory `./hello-world/baseline/1/work`

stdout will be placed at `./hello-world/baseline/1/stdout.txt`

To have ducttape write a bash script that runs this step to

`./hello-world/baseline/commands.sh`, use:

```
$ ducttape -dry-run -write-scripts basic.tape
```

```
task hello_world {  
    echo hi  
    echo >&2 hello  
}
```

The ducttape-structure global is a special directive to ducttape (ignore it for now)

```
global {  
    ducttape_structure=flat  
}
```

2.2 Basics: Writing output files

* Ducttape will assign the paths for the output files

as something like `./hello-world-2/x` and `./hello-world-2/y.txt`

The environment variables `x` and `y-txt` will be set by

ducttape before calling bash

Note that bash disallows variables containing `.`

```
task hello_world_2 > x y_txt {  
    echo writing files $x and $y_txt...  
    echo hello > $x  
    echo world > $y_txt  
}
```

Since variables with dots are not allowed by bash (and therefore ducttape), ducttape allows you to specify the output file name

This is useful so that:

types can still be automatically detected based on extension

it's easy to refer to outputs of programs with hardcoded output names (e.g. `unix split`)

```
task named_output < x=/etc/passwd > named=x.gz {  
    cat $x | gzip > $named  
}
```

```
global {
    ducttape_structure=flat
}
```

2.3 Basics: Reading input files

* This task takes 2 input files (a and b) and produces no output

Ducttape will set the environment variables in \$a and \$b before invoking bash

```
task hello_world_3 < a=/etc/passwd b=/etc/hosts {
    echo "I will be reading files called $a and $b and doing nothing useful with them"
    cat $a>/dev/null
    cat $b>/dev/null
}
global {
    ducttape_structure=flat
}
```

2.4 Basics: Running tasks with dependencies

First a step we already know about...

```
task first > x {
    for i in {1..10}; do
        echo $i >> $x
    done
}
```

* We use first's output "x" as the input "a" by using the = operator

Instead of specifying an absolute path as before, we specify

it as a dependency using the "\$" prefix

```
task and_then < a=$x@first > x {
    cat < $a > $x
}
global {
    ducttape_structure=flat
}
```

2.5 Basics: Using parameters

* Parameters are for variables that aren't files

They are listed after inputs and outputs, using a double colon

Because we distinguish files from parameters, ducttape can check if input files exist before running length commands or submitting jobs to a scheduler

```
task param_step < in=/etc/passwd > out :: N=5 {
    echo "$in has $(wc -l < $in) lines"
    echo "The parameter N is $N"
    echo $N > $out
}
```

* The distinction between files and parameters also means that parameters don't introduce temporal dependencies when they are references (like this step)
"no-dep" can start running in parallel with "param-step"

```
task no_dep :: X=$N@param_step {
    #echo "X=$N" # a bug! this would be caught by ducttape's static analysis of bash
    echo "X=$X"
}
global {
    ducttape_structure=flat
}
```

2.6 Basics: External config files

External config files allow you to separate variables (and even tasks) that don't belong in a more generic workflow.
For example, if you wish to build a workflow that builds a large system based on data, it's possible to separate the main building workflow from any references to specific data files.

```
task hello_external_world :: who=$someone {
    echo hello $who
}
global {
    ducttape_structure=flat
}
```

2.7 Basics: Global variables

Global variables allow you to define input files and parameters that are reused throughout the workflow.

```
task hello_global_world :: who=$someone {
    echo hello $who
}
```

a block of global variables
these are variables that should
be shared among *all* configs

```
global {
    someone=world
    ducttape_structure=flat
}
```

2.8 Basics: Shorthand variable references

```
task hello :: foo=@ {
    echo ${foo}
}
task prev :: a=42 {
    echo ${a}
}
task next :: a=@prev {
    echo ${a}
}
global {
    foo="hello, world"
}
```

3 Packages

3.1 Packages: Understanding the git versioner

* During R&D, software often changes while a workflow is running
To reproduce a workflow, you need to know what version of
the software you ran
in, out, and N are shown only to illustrate syntax
note: the package syntax is currently experimental since minor changes are planned for a future release.

```
global {
    ducttape_experimental_packages=enable
}
task lunchtime : lunchpy {
    $lunchpy/lunch.py Indian Mexican Italian
}
```

* Build commands are only called when versioner indicates a new version

```
package lunchpy :: .versioner=git .repo="git://github.com/mjdenkowski/lunchpy.git" .ref=HEAD {
    # We don't actually need to compile anything for python code,
    # but for the sake of example we'll make this program run a bit faster
    python -m compileall .
}
```

The following implementation of a git versioner is actually built-in and
automatically available to all workflows – it is provided here for clarity

Checkout is run in a sanboxed directory and \$dir will be a subdirectory (makes using git easier)
All other commands are run inside \$dir

As we will see with inline branches, specializations such as checkout and update
inherit all of their parent's parameters so that update has visibility of \$repo and \$rev
repo-version should return an **absolute** revision identifier. This identifier should include any branch information,
if relevant.

local-version should return current **absolute** revision relative to the repository from
which the tool was checked out. This version should **never** be a relative, mutable

revision identifier such as HEAD or TRUNK.

```
versioner git :: repo ref {
  action checkout > dir {
    git clone $repo $dir
  }
  action repo_version > version {
    git ls-remote $repo $ref | cut -f1 > $version
  }
  # Used to confirm version after checkout
  action local_version > version date {
    git rev-parse HEAD > $version
    git log -1 | awk '/^Date/{$1=""; print}' > $date
  }
}
```

3.2 Adapting for Your Environment

Currently, system specific modifications such as where prefix, etc is located is recommended to be done by copying the build scripts. This may seem bad, but build system and repository system already abstract fairly complicated operations. Each package should generally contain small scripts, which act more or less like configuration files. However, it is considered best practice to have variables for such things on each package block rather than buried in the commands. For example, you should make it easy to modify the number of cores used in the build for make -j, scons -j, bjam -j, etc.

4 HyperWorkflows

4.1 HyperWorkflows: What are HyperWorkflows?

Experimentation often requires one to run similar sequences of tasks in a variety of configurations. To compactly represent such workflows, ducttape provides HyperWorkflows.

For example, if we want to determine the effect of data size on some task, we could run the task several times, using input files of differing sizes, as below.

This step shows the simplest way to pack (i.e. create a hyperworkflow) Parentheses with "fileName=(packName: a=x, b=y)" indicates that we will have branches "a" and "b" for file "f" and the name of the Branch Point (aka Pack) will be "packName".

The file "x" will be used for branch a and the file "y" will be used when the branch b is active.

This task will result in two output directories:

```
./has-branches/baseline/1/work
./has-branches/bigger/1/work
```

Because a is the first branch, it is considered the Baseline branch. Therefore, it is given the special directory name "baseline". This will become important later.

Because there is a task with more than one Branch, we say that this workflow has more than one Realization (which can be thought of as a single DAG derivation of this HyperDAG).

Relative paths *as inputs* are resolved relative to this .tape file.

(Remember, relative paths as outputs are resolved relative to the task's working directory)

```
task has_branches < in=(WhichSize: smaller=small.txt bigger=big.txt) > out {  
  cat < $in > $out  
}
```

Since its parent task has several branches (smaller and bigger),
this task will be run once for each parent branch.

This task will also result in two output directories:

./parent-has-branches/baseline/1/work
./parent-has-branches/bigger/1/work

```
task parent_has_branches < in=$out@has_branches {  
  wc -l $in  
}
```

4.2 HyperWorkflows: The default one off plan

Most HyperWorkflows involve more than one branch point. With a large enough number of these, running the cross-product of all branches quickly becomes infeasible.

To address this problem, ducttape provides "plans". A plan is a set of branches that the user wishes to be run at a specific task.

By default, ducttape will execute all tasks, but only for realizations that contain no more than one *non-baseline* branch.

In this example, we will see how to run these variations as "one off" experiments.

We start with a task much like the previous example

```
task one_off_1 < in=(WhichSize: smaller=small.txt bigger=big.txt) > out {  
  cat < $in > $out  
}
```

And now add a child task that also has a packed parameter with multiple branches
a subdirectory will be created under the task directory for one-off-2
for each realization: one-smaller, one-bigger, two-smaller, two-bigger
Notice that the realization directory name is formed by first sorting
the active branches by their branch point names and then removing
any "baseline" branches

```
task one_off_2 < in=$out@one_off_1 :: N=(N: one=1 two=2) {  
  head -n $N < $in  
}
```

By default, ducttape will run each of these as "one off" experiments: Ducttape runs each workflow to completion (still sharing as much substructure as possible) using each branch of every Branch Point, but using the baseline branch for all other Packs. With the default One Off strategy, the following 4 directories will result from running this HyperWorkflow:

./one-off-1/Baseline.baseline/1
./one-off-1/WhichSize.bigger/1
./one-off-2/Baseline.baseline/1 (with the baseline branch "small" as one-off-1/in)
./one-off-2/N.two/1 (with the baseline branch "small" as one-off-1/in)

4.3 HyperWorkflows: Sequence Branch Points

Branches can also be created based on re-running the task multiple times

This can be useful when the underlying task makes calls to a random number generator (e.g. optimizers)

The step receives the integer ID of this trial as a parameter

Ducttape handles the assignment of sequential ID numbers to whichTrial by the 1..10 construction

```
task run_several_times > x :: trial=(WhichTrial: 1..10) {  
  # Use bash's built-in random number generator  
  echo $RANDOM > $x  
}
```

4.4 HyperWorkflows: Custom realization plans

We start with the same example from part 2

```
task planned_1 < in=(WhichSize: smaller=small.txt bigger=big.txt) > out {  
  cat < $in > $out  
}
```

And add a sequence to this step

```
task planned_2 < in=$out@planned_1 :: N=(N: one=1 two=2) M=(M: 1..10) {  
  head -n $N < $in \  
    | head -n $M  
}
```

Defining work plans to get more than just one-off experiments (this is intended primarily for config files)

ducttape can be called with "ducttape -P Basics" to run cross products of branches by adding lines

like the following to config files:

```
plan Basics {  
  # 4 experiments/realizations: the branches one and two with with all branches of whichSize  
  # The * operator indicates a cross-product  
  # The "score" indicates that score is the goal task ("target" in GNU Make lingo)  
  # and only dependencies of that task should be run  
  reach planned_2 via (WhichSize: smaller) * (N: one two) * (M: 1..10)  
  # * 2 experiments/realizations: just the large model (e.g. if it takes much longer to run)  
  # * "planned_1 and planned_2" indicates that those 2 tasks are the goal tasks  
  # This is used only to demonstrate syntax. Since planned_1 is a parent of planned_2,  
  # mentioning it has no effect in this case  
  reach planned_1, planned_2 via (WhichSize: bigger) * (N: one) * (M: 2 8)  
  # It is also possible to use the plan-glob operator '*' as  
  # a branch name to indicate "all branches" should be used in the cross-product  
  # This will result in the realizations:  
  # * WhichSize.bigger (N.one and M.1 are omitted from the name since it is the baseline)  
  # * WhichSize.bigger-N.two (M.1 is omitted from the name since it is the baseline)  
  reach planned_2 via (WhichSize: bigger) * (N: *) * (M: 1)  
}
```

4.5 HyperWorkflows: Branch Grating

Branch points allow you to run a single task and all of its dependents under a particular set of conditions (the realization). But what if you want some dependents to only use a specific configuration? For this, ducttape provides branch grafts.

A branch graft constrains an input or parameter to use only realizations that contain the specified branch(es) – this of course requires that the branch has been defined by the parent task or one of its dependencies. The task at which the graft was requested will then have no further visibility of that branch point – nor will any dependents of that task have visibility of the branch point.

Note: You can graft multiple branches (from different branch points) using comma we don't show an example of this for brevity.

Note: One input may use a graft and while another input does not use a graft. We don't show this usage here.

A use case from natural language processing for branch grafts:

We have a tokenizer that we'd like to run on the training, dev, and test set and then later using exactly one branch from that branch point

```
task preproc < in=(DataSet: train=big.txt test=small.txt) > out {
  cat < $in > $out
}
task trainer < in=$out@preproc[DataSet:train] > model {
  cat < $in > $model
}
task tester < in=$out@preproc[DataSet:test] > hyps {
  cat < $in > $hyps
}
```

4.6 HyperWorkflows: Nested Branch Points

```
task uses_nested < file=$ref_test :: N=$num_refs {
  echo "I will be reading file $file with $N refs"
  head -n $N $file
}
global {
  # Adding multiple branches from a config file
  num_refs=(
    Test:
    baseline=(
      NumRefs:
      oneRef=1
      multiRef=4
    )
    mt02=(
      NumRefs:
      oneRef=1
      multiRef=16
    )
  )
  ref_test=(
    Test:
    baseline=(
```

```

    NumRefs:
    oneRef=small.txt
    multiRef=small.txt
  )
  mt02=(
    NumRefs:
    oneRef=big.txt
    multiRef=big.txt
  )
)
}

```

4.7 Running Multiple Instances (Processes) of Ducttape

Ducttape supports running processs of itself for the same workflow and config. It uses file-based locking to ensure that the processes don't touch the same task at the same time. Ducttape ensures that these workflows will not deadlock each other.

For example, if process A of the workflow starts running first, and then process B starts running later, process B will only rerun any tasks that were planned by process A iff they failed; otherwise, process B will reuse the output generated by process B. If you desire to invalidate the output of process A, you must first kill process A.

However, if there are 3 or more processs of ducttape running at the same time, Ducttape does not guarantee which process will begin running the task first. For example, say process A starts first, and processs B and C start later. Then if some task T fails, either process B or process C will non-deterministically take it over and begin running it. For this reason, this functionality of having multiple processs is primarily useful for 2 purposes:

Adding additional realizations to the workflow's plan without modifying the workflow Fixing outright bugs in tasks that will definitely cause certain tasks to fail. This allows the second process to continue running these doomed tasks and their dependents to complete the workflow.

5 Submitters

5.1 Submitters: Shell

Using the shell submitter is equivalent to directly typing each tasks' commands on the command line

```

task hello_shell {
  echo hello
}

```

\$COMMANDS are the bash commands from some task
In this case, the variable will contain "echo hello"

```

submitter shell :: COMMANDS {
  action run > exit_code {
    eval $COMMANDS
  }
}

```

5.2 Submitters: Sun Grid Engine (Simple)

note: submitter syntax is currently experimental since a major change is planned for a future release

```

global {
    ducttape_experimental_submitters=enable
}
task hello_sge :: .submitter=sge .walltime="00:01:00" .vmem=1g .q=all.q {
    echo hello
}

```

cpus, vmem, walltime, q: these can be passed as parameters to each task: .cpus .vmem .walltime .q
 COMMANDS: the bash commands from some task
 TASK, REALIZATION, CONFIGURATION: variables passed by ducttape

```

submitter sge :: vmem walltime q
                :: COMMANDS
                :: TASK REALIZATION CONFIGURATION {
action run {
    wrapper="ducttape_job.sh"
    echo "$$ -S /bin/bash" >> $wrapper
    echo "$$ -q $q" >> $wrapper
    echo "$$ -l h_rt=$walltime" >> $wrapper
    echo "$$ -j y" >> $wrapper
    echo "$$ -o localhost:$PWD/job.out" >> $wrapper
    echo "$$ -N $CONFIGURATION-$TASK-$REALIZATION" >> $wrapper
    # Bash flags aren't necessarily passed into the scheduler
    # so we must re-initialize them
    echo "set -e # stop on errors" >> $wrapper
    echo "set -o pipefail # stop on pipeline errors" >> $wrapper
    echo "set -u # stop on undeclared variables" >> $wrapper
    echo "set -x # show each command as it is executed" >> $wrapper
    # The current working directory will also be changed by most schedulers
    echo "cd $PWD" >> $wrapper
    echo "$COMMANDS" >> $wrapper
    # Use SGE's -sync option to prevent qsub from immediately returning
    qsub -sync y $wrapper
}
}

```

5.3 Appendix A: Directory Structure

5.4 Destination Directory

\$DEST is by default the \$PWD from which you run ducttape This may become configurable at some point.
 You can also define inside your config the value "ducttape.dest - dir=/my/directory" as the destination directory.

5.5 Packages

Packages get built in \$DEST/ducttape.packages/\$PACKAGE - NAME/\$PACKAGE - VERSION

5.6 The Attic

Partial output or invalidated output from previous runs gets moved to \$DEST/ducttape.attic/\$TASK - NAME/\$REALIZATION - NAME/\$WORKFLOW - VERSION. This may help prevent accidental deletion of data and can help diagnose issues if tasks fail repeatedly. The workflow version is incremented every time the workflow is executed.

5.7 "Flat" Structure

If the "ducttape.structure=flat" directive is in your configuration, tasks will be stored in \$DEST/\$TASK - NAME. We refer to this directory as \$TASK.

The bash code in your task block will start off with its cwd as \$TASK and its output files including ducttape - stdout.txt, ducttape - exit - code.txt, and ducttape - version.txt, ducttape - submitter.sh will be written there. If you re-run the workflow, any partial output may be moved to The Attic (see above).

If you later choose to switch to "ducttape.structure=hyper", you will need to follow the special instructions below.

5.8 "Hyper" Structure

If you do not specify the structure or "ducttape.structure=hyper" is in your configuration, tasks will be stored in \$DEST/\$TASK - NAME/\$REALIZATION - NAME. We refer to this directory as \$TASK.

Just as in the flat structure, the bash code in your task block will start off with its cwd as \$TASK and its output files including ducttape - stdout.txt, ducttape - exit - code.txt, and ducttape - version.txt, ducttape - submitter.sh will be written there. If you re-run the workflow, any partial output may be moved to The Attic (see above).

5.9 Config Directories

A basic workflow without any configs (the configuration blocks using the "config" keyword") or an anonymous config such as "config " will have the the structure defined by flat or hyper above. However, each config is housed in its own top-level directory under which all tasks live.

For the "flat" structure, this makes the \$TASK directory: \$DEST/\$CONF - NAME/\$TASK - NAME

For the "hyper" structure, this makes the \$TASK directory: \$DEST/\$CONF - NAME/\$TASK - NAME/\$REALIZATION - NAME

5.10 Transitioning from "Flat" to "Hyper"

If you started with a flat structure and want to use hyper, ducttape will detect that you changed your structure preference and warn you that the directory structures are incompatible. You'll need to add a special transitional directory as the destination of hyperworkflow tasks, so set "ducttape.transitional - dir=/my/director/hyperdir". We will refer to this directory as \$TRANS. Your original flat tasks won't be moved, but all new tasks will be placed in \$TRANS.

For the new hyper structure, this makes the \$TASK directory: \$TRANS/\$TASK - NAME/\$REALIZATION - NAME

Or if a conf name is specified: \$TRANS/\$CONF - NAME/\$TASK - NAME/\$REALIZATION - NAME

Your original flat tasks will be symlinked into \$TRANS as: \$TRANS/\$TASK - NAME/baseline - ↵ \$DEST/\$TASK - NAME, since each flat task represents the baseline branch of the Baseline branch point in a hyperworkflow.

5.11 What directives are available in Ducttape?

* ducttape - output: Specify the output directory for your workflow ducttape - unused - vars: What should we do when we detect an unused variable? Values: ignore—warn—error ducttape - undeclared - vars: What should we do when we detect an undeclared variable? Values: ignore—warn—error ducttape - experimental - imports: Enable imports. Syntax and semantics are subject to change. ducttape - experimental - packages: We are planning on making minor changes to the package syntax in a future release. ducttape - experimental - submitters: We are planning on making significant changes to the submitter syntax in a future release.

5.12 Why are tasks more like Make targets than functions?

Because you typically only use them once, even over a large number of experimental comparisons (thanks to branch points). Really, you should be thinking about bash scripts and their arguments as the basic unit of reusable functions in ducttape.

Coming soon: A nice big example.

5.13 Job control

Q. What if I want to change the structure of a workflow while it is running? A. No problem. Any tasks that are currently running will always continue to run. All other tasks defined in the previous version of the workflow, but not defined by the new version of the workflow will be cancelled by default (this can be overridden – it might be useful to override if you want to submit several different sets of one-off experiments). Tasks that were defined in the old version of the workflow and are now redefined by new version of the workflow workflow will replace their old definitions (therefore, the old version of the tasks and all of its dependencies will be cancelled).

Why? Ducttape workflows are submitted to an always-running daemon. There is always one daemon per machine, and you are responsible for making sure you submit jobs from one head node per cluster (ducttape makes some attempts to complain if multiple daemons are pointed at the same directory on a shared filesystem).

Q. What if I want to change the input files to a workflow while it is running? A. Ducttape can't guarantee the success of doing this while the workflow is running. A currently-executing step might be using the file you intend to replace. If you're confident this is not the case, go for it. But we warned. See below for details on switching out input files.

Q. What if I want to change the input files when a workflow is partially completed / between runs? A. Just as ducttape manages the versions of the software packages in your workflow, it also checks if file versions differ (via a SHA1 or other heuristics). If they differ, you have 2 options: 1) Instruct ducttape to invalidate and rerun all tasks (and dependents) that use the changed input or 2) Instruct ducttape to ignore the change. In the latter case, ducttape will remember the change and keep a note of it (in case you notice strange inconsistencies later).

TODO: Make example inputs workflows that are part of the tutorial instead of just throwing words at this issue. (These are then unit tests as well).