Dude Documentation

Release 3.1

Diogo Becker

November 13, 2015

CHAPTER 1

Overview

The Dude is a framework to generate configurations, execute experiments, and process results into summaries. An *experiment* is the run of a program given input files, arguments, etc, and resulting in some output on the standard output, in a file, or even in a database. Dude uses as feed a *Dudefile*, which describes a complete set of experiments. Based on this file, Dude creates command lines for each experiment and, if necessary, configuration files of any complexity.

1.1 Optspaces and optpts

A set of experiments is described by an option space, *optspace* for short, and each experiment by an option point in this space, or simply *optpt*. An example of an *optspace* for some arbitrary program is the following:

```
an_optspace = {
  'buffer_size' : [1024, 2048, 4096],
  'timeout'     : [10, 50, 100]
}
```

Each dimension of the optspace has a domain: `[1024, 2048]`, `[10, 50, 100]`, `["high.dat", "low.dat"]`, etc. An optpt in this space is for example:

```
some_optpt = {
  'buffer_size' : 1024,
  'timeout'     : 50
}
```

1.2 From optpts to command lines

The most basic usage of Dude is the creation of command lines based on optpts. For that, the user provides in the Dudefile a simple method called `cmdl_exp` which transforms an optpt into a string. For example, our program is `echo` and simply print on the screen the buffer size:

```
def cmdl_exp(optpt):
    return "echo buffer_size=%d" % optpt['buffer_size']
```

Alternatively, you can define `fork_exp(optpt)` *instead* of `cmdl_exp(optpt)`. If `fork_exp` is defined, Dude will fork its execution and consider the forked process as the experiment. See *Spawn or fork* section for more details on these alternatives, and see *Remote experiments* for information on how to start experiments remotely.

When Dude is invoked, it generates all optpts of an optspace by calculating its cartesian product. It then creates for each experiment a folder, and spawns the command line inside this folder. A minimal Dudefile follows:

```
from dude.defaults import *
dude_version = 3

optspace = {
    'buffer_size' : [1024, 2048, 4096],
    'timeout'     : [10, 50, 100]
}

def cmdl_exp(optpt):
  return "echo buffer_size=%d timeout=%d" %\
    (optpt['buffer_size'], optpt['timeout'])
```

Hint: Dude allows the user to define *Constraints* to limit the optpts to a subset of the optspace.

1.3 Calling the Dude

dude is a command line tool which accepts several commands to start, stop, delete, filter, and aggregate experiments. It is usually started in a folder where there is a Dudefile (see `examples/echo` for an example). The command `info` shows an overview of the Dudefile in the current folder:

```
examples/echo$ ls
Dudefile
examples/echo$ dude info
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Experiment set: examples/echo
--------------------------------------------------------------------------------
Option space:
             buffer_size = [1024, 2048, 4096]
                 timeout = [10, 50, 100]
Experiments: complete space
Summaries  : ['default']
Timeout    : None
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

The typical workflow of a user consists of four steps, two of them performed by Dude:

1. Creation of a Dudefile;
2. Execution of experiments upon invocation of `dude run`;
3. Aggregation of results upon invocation of `dude sum`;
4. Use of resulting aggregations for further plotting and analysis.

1.3.1 Execution

To start executing experiments, Dude is invoked from the command line with a `run` argument in any folder where a Dudefile exists:

```
examples/echo$ dude run

...
```

Dude executes the experiments in time and space isolation. Experiments are started sequentially by Dude, hence, avoiding contention on resources such as network adapters, CPUs, etc. Additionally, an experiment can write and read from its working directory without interfering or being interfered by other experiments. When first started, Dude creates a `raw` subfolder and for each experiment a subfolder in `raw`, for example `raw/exp__buffer_size1024__timeout50`. The latter are called *expfolders*. The working directory of the experiments are always their expfolders.

Once an experiment is finished, either correctly or by crashing, its results on the standard output and its return value are stored in the files `dude.output` and `dude.status` respectively, both placed in the experiment's expfolder. For checking which experiments failed, one can simply type:

```
examples/echo$ dude failed
raw/exp__buffer_size1024__timeout100/dude.output
```

In this example, the experiment with optpt `{ 'buffer_size' : 1024, 'timeout' : 100 }` returned with a value different than 0 (it failed). When invoking `dude run` again, only failed (or not yet run) experiments are executed. Dude provides several other commands to manage expfolders (see TBD).

Hint: See *Execution order* to learn how to specify the execution order of the experiments.

1.3.2 Summaries

Dude can collect, filter, aggregate any information from experiments with *Summaries details*. For that the user invokes

```
examples/echo$ dude sum
```

By default, Dude simply concatenates the output to the stdout of every experiment into the file `output/default`. After calling `dude sum`, the user can access the resulting aggregation file with any program to further process, analyze or plot it, for example:

```
examples/echo$ cat output/default

1024 10 buffer_size=1024 timeout=10
1024 50 buffer_size=1024 timeout=50
1024 100 buffer_size=1024 timeout=100
2048 10 buffer_size=2048 timeout=10
2048 50 buffer_size=2048 timeout=50
2048 100 buffer_size=2048 timeout=100
4096 10 buffer_size=4096 timeout=10
4096 50 buffer_size=4096 timeout=50
4096 100 buffer_size=4096 timeout=100
```

Dude provides several *summary* classes which can be added directly to the Dudefile as follows:

```
import dude.summaries
summaries = [ dude.summaries.LineSelect('stdout') ]
```

Additionally, the user can extend any of the summaries and add it to the `summaries` variable in the Dudefile.

1.4 From optpts to configuration files

Dude can generate configuration files before executing the command line returned by `cmdl_exp`. For that the user has to provide an *Prepare and finish* method, which is invoked inside the experiment's folder. Here is an example:

```
dude_version = 3
from dude.defaults import *

```

```
optspace = {
    'buffer_size' : [1024, 2048, 4096],
    'timeout'     : [10, 50, 100]
}

def prepare_exp(optpt):
  f = open("config.txt","w")
  print >>f, "buffer_size=%d timeout=%d" %\
    (optpt['buffer_size'], optpt['timeout'])
  f.close()

def cmdl_exp(optpt):
  return "cat config.txt"
```

Because Dude runs the experiments in separate expfolders, `prepare_exp` do not overwrite the configuration files of other experiments even if they are named in the same way for all experiments.

CHAPTER 2

Dude command-line

2.1 Information and basic commands

info
Show informations about the experiments.

list
List all expfolders.

run
Run experiments (only those that haven't been run or that have failed).

sum
Summarize experiments that successfully ran.

failed
Show all experiments that failed (returned code different than 0).

missing
Show all experiments that haven't been executed or have failed.

-f <FILE>, **--file** <FILE>
Read FILE as a Dudefile. Default `FILE=Dudefile`.

2.2 Filtering experiments

-y, --filter-inline
Select experiments using inline filters separated by semicolons:

```
dude run/list/sum -y "option1=value;option2=[value3,value4]"
```

-p, --filter-path
Select experiments starting with PATH:

```
dude run/list/sum -p raw/exp__optionXvalY
```

-i, --invert-filters
Invert filter selection. Example:

```
dude run/list/sum -i -y "option1=value;option2=[value3,value4]"
```

2.3 Run specific options

-o
Show experiment's output.

--force
Force execution. For example:

```
dude run --force -y "option1=value;option2=[value3,value4]"
```

--skip-global
Skip global prepare.

--global-only
Run global prepare only.

2.4 Sum specific options

--ignore-status
Include failed experiments in the summaries.

-b, --backend
Default backend is file.

Backend to use for summary: sqlite3, json, and file.

2.5 List specific options

-d, --dict
Show output in dict format.

2.6 Advanced commands and options

run-once
Creates a folder called "once" and runs one experiment inside. The selection with -y or -x or -p is necessary such that only one experiments is selected.

Advanced filtering can be achieved with the following command-line options.

-x `<filter>`, **--filter** `<filter>`, **--select** `<filter>`
Select experiments using filters written in Dudefile. One can use multiple filters, for example:

```
dude run/list/sum -x filter1,filter2
```

The filter has to be implemented inside the Dudefile. (More details TBD)

-a, --args
Give arguments to Dudefile separated by semicolons:

```
dude run/list/sum -a "option1=value;option2=[value3,value4]"
```

Overall structure

3.1 Optspaces and optpts

Each experiment has a *configuration*, which is the set of all controllable variables of the experiment, e.g., buffer size of 1024 bytes and timeout of 50 seconds. The configuration might be "materialized" in a configuration file, in a command line, etc. For Dude, a configuration is called an option point, or *optpt* for short, for example:

```
some_optpt = {
  'buffer_size' : 1024,
  'timeout'     : 50
}
```

Each dimension of an optpt has a domain: `[1024, 2048]`, `[10, 50, 100]`, `["high.dat", "low.dat"]`, etc. The n-dimensional space of all option dimensions and their domains is called options space, or *optspace* for short. For example:

```
an_optspace = {
  'buffer_size' : [1024, 2048, 4096],
  'timeout'     : [10, 50, 100]
}
```

3.2 Constraints

By default, Dude generates all the optpts of an optspace by calculating its cartesian product. Parts of the optspace can be however removed by using *constraints*. A constraint is a *function* that takes an optpt as parameter and returns `True` if the optpt is valid otherwise `False`. For example, assume that performing an experiment with `buffer_size` of 4kB only makes sense if the `timeout` is at least 50. This constraint can be expressed as the following function:

```
def my_constraint(optpt):
  if optpt['buffer_size'] == 4096:
    return optpt['timeout'] >= 50
  else:
    return True
```

3.3 Directory structure and runs

Dude executes the experiments in time and space isolation. It starts experiments sequentially, hence, avoiding contention on resources such as network adapters, CPUs, etc. .. To start multiple experiments concurrently, multiple instances of Dude can be started. Additionally, Dude starts each experiment with a different *expfolder* as working directory. An experiment can write and read from its working directory without interfering or being interfered by other experiments. When first started, Dude creates a `raw` subfolder and for each experiment a subfolder in `raw`, for example `raw/exp__buffer_size1024__timeout50`. The latter are called *expfolders*. While an optpt represents the configuration of an experiment, the expfolder is the *results* of an experiment.

Once an experiment is finished, either correctly or by crashing, its results on the standard output and its return value are stored in the files `dude.output` and `dude.status` respectively, both placed in the experiment's expfolder.

Important: It is recommended that any other file the experiment generates to be stored in the process' working directory, which is always the expfolder. If that is not the case, Dude provides a method to be called after the experiment, which can be used to copy the desired results to the expfolder.

CHAPTER 4

Run details

4.1 Spawn or fork

There are two types of experiments that Dude can start: command lines or forked process. You can define only one of them in your Dudefile. Command-line experiments are executed if the `cmdl_exp(optpt)` method is defined. `cmdl_exp` must return the command line to be executed as a string, for example:

```
def cmdl_exp(optpt):
  return "echo buffer_size=%d" % optpt['buffer_size']
```

In this example, when `dude run` is issued on the terminal, the command line `echo buffer_size=X` will be called for every value of X defined in the `buffer_size` dimension of the optspace.

Forked-process experiments are executed if the `fork_exp(optpt)` method is defined. There are two use cases for `fork_exp`. First, for experiments that are very simple such that they can be expressed as a couple of python lines of code, for example:

```
def fork_exp(optpt):
  r = is_prime(optpt['number'])
  if r: print r, "is prime"
  else: print r, "is not prime"
```

The second use case is for very complex which need several commands, possibly running in parallel. For example, let's assume we want to experiment with a client and a server program varying parameters of both programs:

```
import subprocess
def fork_exp(optpt):
  s = subprocess.Popen('my_server --buffer=%d' %\
                       optpt['server_buffer_size'],
                       stdout = sys.stdout, stderr = sys.stderr)

  c = subprocess.Popen('my_client --buffer=%d --threads=%d --timeout=%d' %\
                       optpt['client_buffer_size'],
                       optpt['client_threads'],
                       optpt['experiment_time'],
                       stdout = sys.stdout, stderr = sys.stderr)

  # Block until client is done.
  rc = c.wait()
  # If the return code is different from 0, there was an error.
  # We can either assert that, or return the error code as return value.
  assert rc == 0, "client failed"
  # Terminate server with SIGTERM. We assume it handles the signal and
  # exits with 0.
  s.terminate()
  rs = s.wait()

  # Return the retcode of the server. In this example if the server or the
  # client failed, the experiment must be repeated/fixed.
  return rs
```

When `dude run` is issued on the terminal, the Dude process is forked just before calling `fork_exp` for every optpt in the optspace. If the `fork_exp` fails with an assertion, throwns an exception, or returns a value different from 0, the experiment is marked as failed.

> **Warning:** subprocesses spawned inside `fork_exp` must have stdout and stderr set to `sys.stdout` and `sys.stderr` respectively. Otherwise, Dude cannot log their output (see *Output logging* for details).

4.2 Timeouts and signals

By default, an experiment can take an unbounded amount of time to terminate. If the user wants to limit this time, a `timeout` variable can be set in the Dudefile (seconds as unit). If the experiment times out, it is killed by Dude. Similarly, if the user presses "ctrl-c", any running experiment is immediately killed.

4.3 Prepare and finish

Dude provides hooks which are executed before and after every experiment. You can use `prepare_exp(optpt)` to create configuration files for your experiment, and `finish_exp(optpt, status)` to perform cleanup actions.

Here is an example:

```
dude_version = 3
from dude.defaults import *

optspace = {
    'buffer_size' : [1024, 2048, 4096],
    'timeout'     : [10, 50, 100]
}

def prepare_exp(optpt):
  f = open("config.txt","w")
  print >>f, "buffer_size=%d timeout=%d" %\
    (optpt['buffer_size'], optpt['timeout'])
  f.close()

def cmdl_exp(optpt):
  return "my_program -f config.txt"

def finish_exp(optpt, status):
  if status == 0:
    cleanup_program_successful()
  else:
    cleanup_program_failed()
```

4.4 Output logging

Dude redirects the output (stdout and stderr) of each experiment to a log file. In particular, `dude.prepare_exp`, `dude.output`, and `dude.finish_exp` log files will be created, corresponding to the different phases of an experiment. See *Run specific options* for information on how to display the output on the screen.

4.5 Execution order

Dude executes experiments in some "random" order decided by the `cmp` built-in Python method. Sometimes it can be useful to define the order in which experiments are going to be executed. Dude provides the optional function `optpt_cmp(optpt1, optpt2)` for that, which can be define in your Dudefile and should implement the `cmp` interface (See cmp documentation for details).

The most common use of this feature is to tell Dude to iterate on the dimensions in some order. For example, we would like test a server and a client application with the following options:

```
optspace = {
  'server_buffer_size' : [1024, 2048, 4096],
  'client_buffer_size' : [1024, 2048, 4096]
}
```

The server takes long to start, so it would be good to test first all client options before restarting the server with another option (see *Spawn or fork* for an example on how to start multiple processes). Our Dudefile could look like this:

```
from dude.defaults import order_dim
optpt_cmp = order_dim(['server_buffer_size', 'client_buffer_size'])

last_server_buffer_size = None
def fork_exp(optpt):
  if last_server_buffer_size != optpt['server_buffer_size']:
    stop_server()
    last_server_buffer_size = optpt['server_buffer_size']
    rc = start_server(optpt)
    assert rc == 0, "server failed on start"

  rc = start_client(optpt)
  # return client's return code
  return rc
```

`order_dim` is a helper function defined in `dude.defaults` that order the dimensions in the list order given as argument. In the example, first all `client_buffer_size` values would be executed before the `server_buffer_size` would change.

Summaries details

Summaries are used to extract information from executions. Summaries are usually used to perform simple aggregations only; tools such as Gnuplot, matplotlib and R can be used for further aggregation, analysis and plotting.

Summaries are added to the `summaries` variables in a Dudefile, for example:

```
import dude.summaries
summaries = [ dude.summaries.LineSelect('my_summary') ]
```

Since version Dude 3.1 summaries inherit a single SummaryBase class. All the following parameters are available in any summary.

class `dude.summaries.`**`SummaryBase`**(*name*, *groupby=[]*, *header=None*, *quiet=False*)
Base class for summaries.

Parameters

- **`name`** – prefix of output filename
- **`groupby`** – groupby dimensions
- **`header`** – columns string separated by spaces
- **`quiet`** – be quiet

Summaries can either process output from the stdout/stderr of a program execution or from files generated during the execution.

5.1 stdout/stderr summaries

LineSelect and MultiLineSelect

class `dude.summaries.`**`LineSelect`**(*name*, *groupby=[]*, *header=None*, *regex='.*'*, *split=None*, *quiet=False*)
Filters and splits lines from stdout/stderr.

Parameters

- **`files`** – a list of filenames or a string with wildcards
- **`regex`** – regex to select lines
- **`split`** – function to split selected lines, if None, *(lambda l: l)*.

class `dude.summaries.`**`MultiLineSelect`**(*name, groupby=[], filters=[('', '.*', <function <lambda> at 0x7fbe54c1c0c8>)], quiet=False*)
Filters and splits lines from stdout with multiple rules

function to split selected lines, if None, *(lambda l: l)*.

Parameters

- **`filters`** – list of (header, regex, split)
- **`fname_split`** – function to split the file names
- **`fname_header`** – header for filename, default = "file"

5.2 file summaries

class `dude.summaries.`**`FilesLineSelect`**(*name*, *files*, *groupby=[]*, *header=None*, *regex='.*'*, *split=None*, *fname_split=None*, *fname_header='fname'*, *quiet=False*)

Filters and splits lines from files:

Parameters

- **`files`** – a list of filenames or a string with wildcards
- **`regex`** – regex to select lines
- **`split`** – function to split selected lines, if None, *(lambda l: l)*.
- **`fname_split`** – function to split the file names
- **`fname_header`** – header for filename, default = "fname"

class `dude.summaries.`**`FilesMultiLineSelect`**(*name*, *files*, *groupby=[]*, *filters=[('', '.*', <function <lambda> at 0x7fbe54c1c230>)]*, *fname_split=<function <lambda>>*, *fname_header='fname'*, *quiet=False*)

Filters and splits lines from selected files with multiple rules

Parameters

- **`files`** – a list of filenames or a string with wildcards
- **`filters`** – (column, regex, split)
- **`fname_split`** – function to split the file names
- **`fname_header`** – header for filename, default = "fname"

5.3 other summaries

class `dude.summaries.`**`JsonSelect`**(*name*, *path*, *filename*, *header*, *groupby=[]*, *quiet=False*)

Selects entries from a Json file.

5.4 Backends

Over time some different backends turned out to be useful. Dude supports file, json and sqlite backends.

CHAPTER 6

Other stuff

6.1 Filters

6.1.1 Inline filters

6.1.2 Complex filters

6.2 Running “once”

6.3 Remote experiments

CHAPTER 7

Indices and tables

- genindex
- modindex
- search