decu Documentation

Release 0.0.1

leotrs

Nov 02, 2017

Documentation

`decu` stands for "Decu is a Experimental Computation Utility". `decu` is a suite of command line tools to automate the menial tasks involved in the development of experimental computation projects.

We define a "experimental computation" script as a script that reads some data, performs an experiment (run an algorithm, train a model, plot a figure, etc), saves some results to disk, and then quits. Some tasks that are usually involved in this are:

1. file I/O: choosing file names for results and plots
2. multiprocessing: running many experiments/scripts in parallel
3. timekeeping: how long algorithms take to run
4. cloud service integration
5. logging

`decu` was born from the realization that none of these tasks have anything to do with (and in fact get in the way of) the actual experimentation being done. Furthermore, experimental computation scripts tend to mix together code dedicated to these tasks with code intended to run the actual algorithms of interest, thus making said algorithms harder to maintain and debug.

The main goal of `decu` is to provide standardization and automation of these tasks, with end code that clearly separates experimental computation from bookkeeping and other auxiliary code.

`decu` is built with Best Practices for Data Science, by Greg Wilson et al, in mind. See 1 and 2.

CHAPTER 1

Quick Start

If you're in a hurry and want to enjoy all the benefits from `decu` without having to go through a two hour long lecture, this is the place for you.

1.1 Installation

Clone this repo, `cd` to the decu directory and do

```
$ pip install .
```

Now you have a local installation of `decu`. If you are going to make edits to `decu`, don't forget to use the `-e` flag.

1.2 Start a new project

To start a new project, choose a directory in your file system, say `root_dir`. Navigate to `root_dir` and execute the following command.

```
$ decu init
```

`decu` will generate the default directory structure under `root_dir`.

Here's the what you need to know in order to use `decu` inside `root_dir`:

- All contents relating to your project (data, code, figures, documentation, manuscripts, etc) should be in subdirectories of `root_dir`.
- All `decu` commands should be invoked from `root_dir`, not any of its subdirectories.

1.3 Development

If you `ls` your `root_dir`, you will find a newly created `scr/` directory. This is where all your computational experiments go. For this example, we will use the following file, `script.py`:

```
import decu

class MyScript(decu.Script):
    @decu.experiment(data_param='data')
    def exp(self, data, param, param2):
        return [x**param + param2 for x in data]

    def main(self):
        data = range(100)
        result = self.exp(data, 2.72, 3.14)
```

Place this file inside `src/` and call the following command.

```
$ decu exec src/script.py
```

`decu` will run your script, leave a log file inside `logs/` and save the results to disk inside `results/`.

Here's the what you need to know in order to develop `decu` scripts:

- A script that is to be run with `decu` must subclass `decu.Script` and override the `main` function. Say you call this subclass `MyScript`.
- Methods of `MyScript` that are intended to run your main algorithms should be decorated with `@decu.experiment`.
- `decu.experiment`-decorated methods must accept at least one parameter. This will be treated as the data input, and its name must be passed to the decorator. In the above example, the `exp` method is decorated with `decu.experiment` with data input parameter *data*, and thus we pass the string `"data"` to `decu.experiment`.
- All other parameters will be treated as experimental parameters that tune the behavior of the algorithm.

1.4 Inspection

In the above example, we called `exp` with `param2 = 3.14`. Say we made a mistake and the correct value for `param2` should have been `6.28`. Instead of fixing the file and running the whole script again (which in a real world scenario could be costly and/or time-consuming), with `decu` we have an alternative.

After the last step, `decu` saved the results of your script in a file inside `results/`. Call the file `<result_file1>`. Now execute:

```
$ decu inspect results/<result_file1>
>>> import decu
>>> import numpy as np
>>> import src.script as script
>>> script = script.MyScript('<root_dir>', 'script')
>>> # loaded result

In[1]:
```

What `decu` just did is initiate an iPython session ready for us to fix our mistake. The contents of `<result_file1>` have been loaded in the variable `result`.

Observe that we can fix our mistake by adding `3.14` to each element in the `result` array. So we do so in iPython:

```
In[1]: result += 3.14
```

Our mistake is now fixed. We need only save the new result file, which we can do manually with `numpy` (observe that `decu` loaded `numpy` for us already):

```
In[2]: np.savetxt('results/fixed.txt', result)
```

Or, if we want to use the same naming scheme as `decu` is using for all files, we can do the following. Note that `decu` also loaded in our iPython session our `script.py` and instantiated `MyScript` as `script`:

```
In[3]: np.savetxt(script.make_result_file("exp", 0), result)
```

We need to specify `"exp"` as this was the `@decu.experiment`-decorated method that originally created the result. We can close iPython now.

Here's what you need to know in order to use `decu inspect`:

- `decu`'s default file names may seem complicated but they are useful when inspecting results, as `decu` can know from the file name what script, class, and method generated a result file.

1.5 We're done!

In the course of this quick tour-de-force, we have a new directory structure for our new project, added a script and fixed a minor mistake by using `decu`'s inspecting capabilities. All bookkeeping, file names, logging, and directories have been handled by `decu` for us.

This is only a sliver of what one can do with `decu`, but it should give you the minimum you need to know to start using it in your projects.

For a more detailed example, see the tutorial.

CHAPTER 2

Tutorial

This is the `decu` tutorial. Follow a long for a tour of `decu`'s main features.

2.1 Example Project

After you have installed `decu` in your (virtual) environment, you can follow along with this example project. This example will showcase the three main sets of functionality that `decu` provides: project organization, bookkeeping, and inspection. It is best if you follow along by copying and pasting the source code we show below in a temporary project.

2.1.1 Project organization: `decu init`

The first step is to get our file system set up to run an experimental computation project. Navigate to an empty directory where your project is going to live. Call this directory `root_dir`. Now execute

```
$ decu init
Initialized empty decu project directory in <root_dir>
```

`decu` follows a somewhat strict template of what a project filesystem should look like, and with `$ decu init`, all we're doing is setting up `root_dir` to reflect `decu`'s organization. If you now `ls` your `root_dir` you will see something like the following.

```
root_dir
    src
    data
    logs
    pics
    results
```

The purpose of each directory created by `decu` is straightforward. The `data` directory is assumed to contain raw or processed data files. The `logs` directory will store the logs of running experiments. The `pics` and `results`

directories will hold plots, figures, and result files generated by your `decu` scripts. The `src` directory will contain said scripts.

2.1.2 Project bookkeeping: `decu exec`

With this directory structure, we can now start coding our computational experiments. Create a file `script.py` inside `root_dir/src` that contains the following.

```
import decu

class MyScript(decu.Script):
    @decu.experiment(data_param='data')
    def exp(self, data, param, param2):
        return [x**param + param2 for x in data]

    def main(self):
        data = range(100)
        result = self.exp(data, 2.72, 3.14)
```

In `script.py` we subclass `decu.Script`, define an experiment method `exp`, and call it on some data from the `main` method. Note that the method is decorated with `@decu.experiment` which requires us to specify which of `exp`'s parameters is treated as the data input. All other parameters are treated as 'experimental parameters'. More on experimental parameters later. That's basically all that this code does.

What could we expect `decu` to do in this simple example? Bookkeeping! Note that we didn't save the results of our experiment to disk, or used log files or `print` calls to document what the script is doing. This was not an oversight, it was on purpose. In fact, `decu` will do all of this for us.

Before running the above script, the directory structure should look as follows.

```
root_dir
    src
        script.py
    data
    logs
    pics
    results
```

> **NOTE**: After having called `$ decu init` in `root_dir`, all successive calls to `decu` should be made from `root_dir` itself. That is, do not `cd` into `src/` and then call `decu`. All console commands from here on will be done from `root_dir`.

Simple run

Now `cd` to `root_dir` again and run our script through `decu`.

```
$ decu exec src/script.py
```

After executing `script.py`, we can now take a look at what happened to `root_dir`.

```
root_dir
    src
        script.py
    data
    logs
        log_file1--0.txt
    pics
    results
        result_file1--0.txt
```

Here, both `log_file1.txt` and `result_file1.txt` will have a name including the date and time of execution and the name of the script that generated these files, among other information.

This is (one of) the main features of `decu`. We needn't specify what information we want to log, or the file name where we want to save our experimental results. In fact, we didn't even need to manually save the results to disk ourselves. `decu` will take care of the bookkeeping.

To see more specifically what `decu` saves to the log file, do

```
$ cat logs/log_file1.txt
[<time>]INFO: Starting exp--0 with {'param': 2.72, 'param2': 3.14}.
[<time>]INFO: Finished exp--0. Took 4e-05.
[<time>]INFO: Wrote results of exp--0 to results/result_file1.txt.
```

Since our script is very simple, `decu` just needed to log three lines. However, you fill find there's a trove of information here. Without writing a single line of logging or I/O code, we now have:

1. a unique file with a time-stamped recount of what our script did,
2. a record of the experimental parameters with which `exp` was called,
3. the time it took to run `exp`,
4. a file containing the results of running `exp` with the recorded parameters. This file contains in its name the name of the script and the function that generated its contents.

To understand why the `log_file1.txt` logs the call to `exp` as `exp--0`, we need to modify `script.py` a little.

Multiple runs

```
import decu

class MyScript(decu.Script):
    @decu.experiment(data_param='data')
    def exp(self, data, param, param2):
        return [x**param + param2 for x in data]

    def main(self):
        data = range(100)
        result = self.exp(data, 2.72, 3.14)
        params = [(data, x, y) for x, y in zip([1, 2, 3], [-1, 0, 1])]
        result2 = decu.run_parallel(self.exp, params)
```

We have included further experiments now. We are calling the same method `exp` but with a different choice of parameters each time. `decu.run_parallel(method, params)` will call `method(*l)` for each element `l` in `params`, and it will do so by using Python's `multiprocessing` library, which means that these experiments will be run in parallel.

To execute this new version, we do

```
$ decu exec src/script.py
```

First of all, take a look at the current state of `root_dir`, after a second run our script.

```
root_dir
    src
        script.py
    data
    logs
        log_file1.txt
        log_file2.txt
    pics
    results
        result_file1--0.txt
        result_file2--0.txt
        result_file2--1.txt
        result_file2--2.txt
        result_file2--3.txt
```

Here's what happened: `decu` created a new log file for this second experimental run, `log_file2.txt`. It also generated one result file for each of the experiments we ran. To understand the contents of the new result files, we need only read the new log file. (Your output maybe slightly different in the order of the lines.)

```
$ cat logs/log_file2.txt
[<time>]INFO: Starting exp--0 with {'param': 2.72, 'param2': 3.14}.
[<time>]INFO: Finished exp--0. Took 0.0004s.
[<time>]INFO: Wrote results of exp--0 to results/2017-10-20 18:21:28.374962--script--
↪exp--0.txt.
[<time>]INFO: Starting exp--2 with {'param': 2, 'param2': 0}.
[<time>]INFO: Starting exp--1 with {'param': 1, 'param2': -1}.
[<time>]INFO: Starting exp--3 with {'param': 3, 'param2': 1}.
[<time>]INFO: Finished exp--2. Took 4e-05s.
[<time>]INFO: Finished exp--1. Took 5e-05s.
[<time>]INFO: Finished exp--3. Took 6e-05s.
[<time>]INFO: Wrote results of exp--2 to results/2017-10-20 18:21:28.374962--script--
↪exp--2.txt.
[<time>]INFO: Wrote results of exp--1 to results/2017-10-20 18:21:28.374962--script--
↪exp--1.txt.
[<time>]INFO: Wrote results of exp--3 to results/2017-10-20 18:21:28.374962--script--
↪exp--3.txt.
```

The first three lines are familiar. They correspond to the call to `exp` that we had before, and they provide similar information. We now know that `result_file2.txt` contains the result of running `exp` with parameters `{'param': 2.72, 'param2': 3.14}`. Then we used `run_parallel` to run `exp` over a list of parameters, and we get the last nine lines, which contain similar information as before, but for the three additional times we called `exp`. Observe that each time we call `exp`, the log file includes two dashes followed by a number, `--x`. This is a way for identifying different calls to the same experiment, and serves to tell which result file belongs to which experiment call. Note that the result files also contain the same identifier at the end. Without this identifier, our result files would all overwrite each other, or else there would be no way to tell which contains the result of which experiment call. In the example above, we need only match the identifier of a result file name with the experiment identifiers in log file, to know

1. the method that generated the result file,
2. the time it took to generate the file, and
3. the experimental parameters that were used to generate it.

In other words, with these identifiers we can cross-reference results and experiment runs by using the log file. The run identifiers are guaranteed to be unique for each call to `exp`, even when using parallelism with `run_parallel`.

Oh, never mind the fact that we just used multiprocessing to run our experiments in parallel, with no additional imports and in a single call of `run_parallel`.

Figures

So now `decu` is handling logging, directory bookkeeping, cross-referencing experimental parameters and results, and parallelism, in 12 lines of python (two of each are empty BTW). But there's more!

Say now that you want to plot a pretty picture from your results. Enter the `@figure` decorator, used in the following version of `script.py`.

```
import decu
import matplotlib.pyplot as plt

class MyScript(decu.Script):
    @decu.experiment(data_param='data')
    def exp(self, data, param, param2):
        return [x**param + param2 for x in data]

    @decu.figure()
    def fig(self, data, results):
        for res in results:
            plt.semilogy(data, res)

    def main(self):
        data = range(100)
        result = self.exp(data, 2.72, 3.14)
        params = [(data, x, y) for x, y in zip([1, 2, 3], [-1, 0, 1])]
        result2 = decu.run_parallel(self.exp, params)
        self.fig(data, result2)
```

After importing `matplotlib`, we have added the `fig` method, which we have decorated with `@decu.figure()`, and we call at the end of `main`.

You know the drill now:

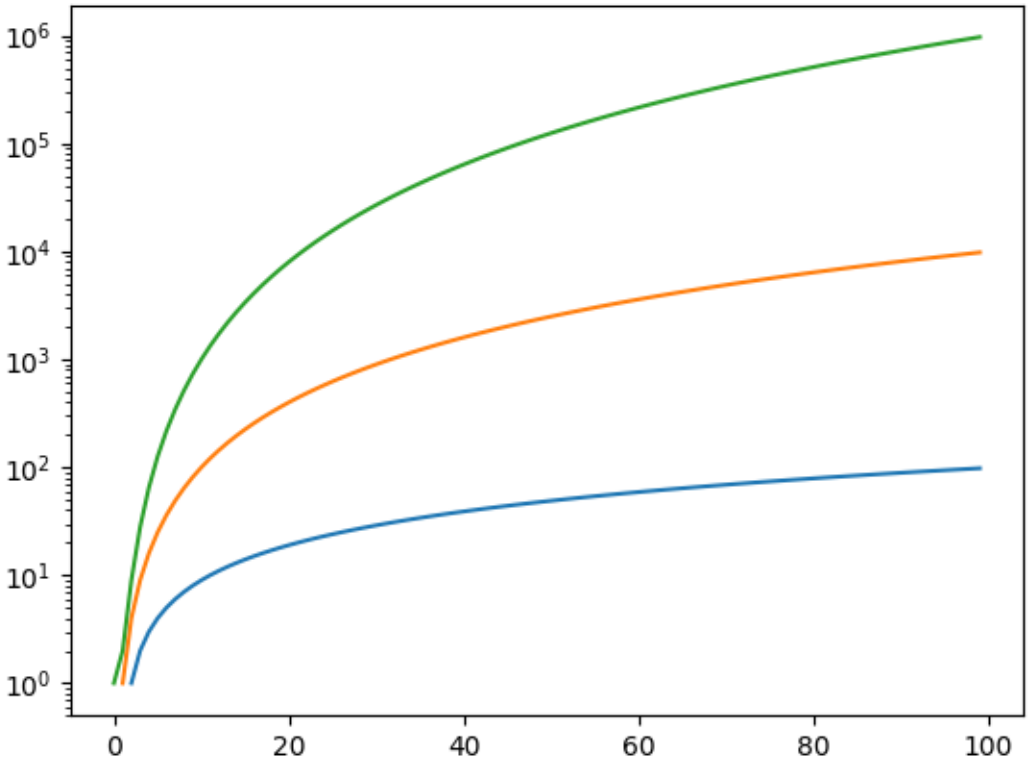
```
$ decu exec src/script.py
```

The `root_dir` should now look as follows.

```
root_dir
    src
        script.py
    data
    logs
        log_file1.txt
        log_file2.txt
        log_file3.txt
    pics
        fig_file1.png
    results
        result_file1--0.txt
        result_file2--0.txt
        result_file2--1.txt
        result_file2--2.txt
        result_file2--3.txt
        result_file3--0.txt
        result_file3--1.txt
```

```
        result_file3--2.txt
        result_file3--3.txt
```

As before, we get our log file `log_file3.txt` and four result files. We also get out plot inside `pics/`. "But wait!"-I hear you say-"we never saved the plot to disk!" Exactly. You can open this file to convince yourself of how wonderful `decu` is.



You can also read the log file to see that it mentions which method (`fig`) generated the new figure file.

2.1.3 Project debugging: `decu inspect`

Oh, darn. We forgot to add a title to our plot. After we have modified our `fig` function to include a nice title, we can generate the new plot in a number of ways. First, we can run the whole thing again. This becomes increasingly cumbersome (and sometimes outright impossible) if `exp` takes too long to run, as it often does in real life. Second, since we have the result file, we can pop into a python interpreter, read the result from disk, and call `fig` again. This would require us to not only load the result, but the file `script.py` and instantiate the class `MyScript`. How tedious.

OR, we can use `decu` to do exactly that.

```
$ decu inspect results/result_file3*
import decu
import numpy as np
import src.script as script
script = script.MyScript('root_dir', 'script')
```

```
# loaded result

In [1]: data = range(100)

In [2]: script.fig(data, result)

In [2]: exit
```

If you have the necessary data in a file, then another possibility is:

```
$ decu inspect result_dict=results/result_file3* --data=data/data_file1.txt
import decu
import numpy as np
import src.script as script
script = script.MyScript('/tmp/root_dir', 'script')
# loaded result
# loaded data

In [1]: script.fig(data, result)

In [2]: exit
```

And in that case, yet another possibility, and the most efficient one, is to use the `-c` flag:

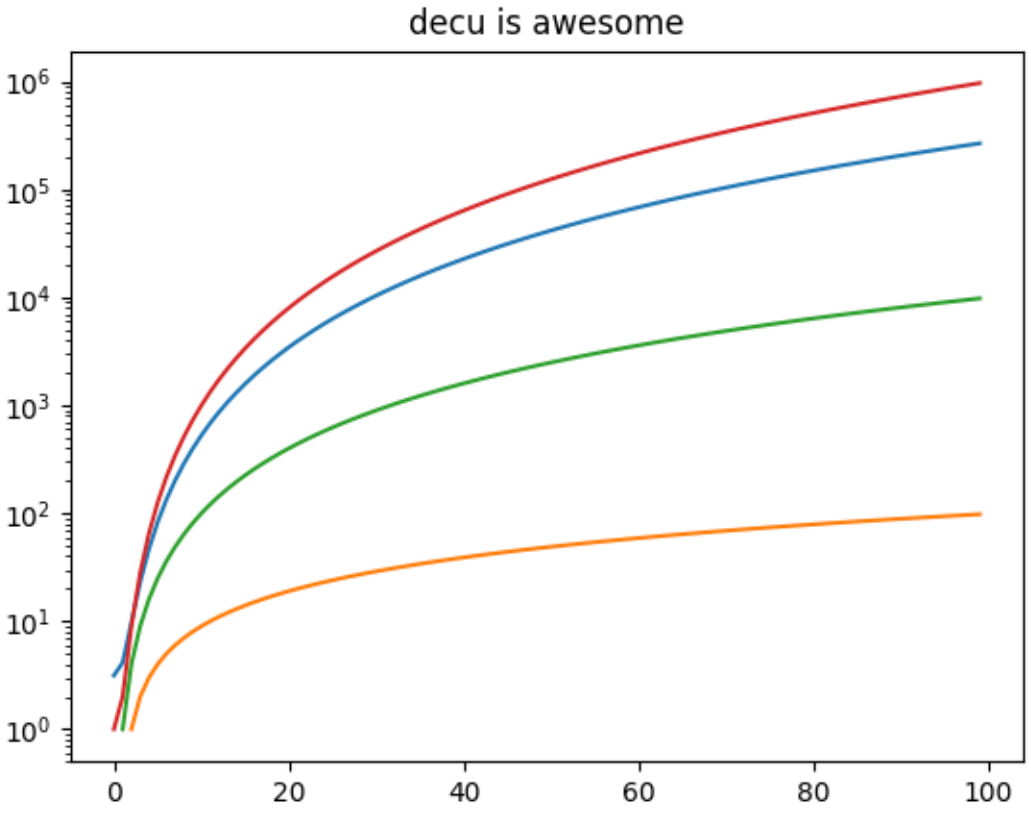
```
$ decu inspect -c "script.fig(data, result)" \
result_dict=results/result_file3* \
--data=data/data_file1.txt

import decu
import numpy as np
import src.script as script
script = script.MyScript('/tmp/root_dir', 'script')
# loaded result
# loaded data
script.fig(data, result)
exit

$
```

The `-c` flag takes an arbitrary string, executes it after the initial file loading is done, and then quits ipython. This makes it possible to fix our figure in a single `decu` call, without having to drop to an interpreter or manually load anything. Nifty, yes?

The final result is:


decu is awesome
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0
20
40
60
80
100

CHAPTER 3

Customizing `decu`

No two projects are the same, and thus `decu` is built with customization and extensibility in mind. For this purpose, `decu` relies heavily on a configuration file accessible by the user through `decu.config`. See the `decu.config` module for more information.

3.1 The configuration file

The configuration file is a standard `.ini` file, usually called `decu.cfg`, and read by the `configparser` standard library. It's broken down into sections pertaining different aspects of the `decu` system. `decu` provides defaults for all options so no customization is necessary before using `decu`. However, if you want to customize the system, all you need to do is create a `decu.cfg` file in your project's root directory and override some options. Which options are available and how to override them is documented in a later section.

3.2 How the config file is read

There are three configuration files that `decu` is aware of. Suppose you are working on a project located at `root_dir`. Then, `decu` reads three different configuration files, in the following order:

1. Global: The default `decu.cfg` configuration file that comes with the `decu` installation. Modify this file only if you want your changes to be system-wide.
2. User-local: `~/.decu.cfg`. Modify this file if you want all your (the user's) projects to see the same changes.
3. Project-local: `root_dir/decu.cfg`. Modify a single project.

Files read later have precedence, so you can always override the global options by supplying a `decu.cfg` under your `root_dir`.

Note: it is not recommended to change options during the development of a project. It is best to customize `decu` before development has started and to stick to one configuration file for the duration of a project. At the moment of writing, it is unclear which options are safe to customize during the course of a project. Thus it is HIGHLY DISCOURAGED

to modify the global configuration file as the changes will take place even in projects that are already in development. It is best to use project-local configuration files for each project.

3.3 Configuration options

Each configuration option is documented in the global configuration file, which we paste here.

CHAPTER 4

Reference

Decu is a library for experimental computation scripts.

4.1 core.py

Main decu classes and decorators.

class decu.core.**Script**(*project_dir=''*, *module=None*)
Bases: object

Base class for experimental computation scripts.

data_dir = 'data/'

figure_fmt = 'png'

figures_dir = 'pics/'

gendata_dir = 'data/'

make_figure_basename(*fig_name*, *suffix=None*)

make_result_basename(*exp_name*, *run*)

results_dir = 'results/'

scripts_dir = 'src/'

decu.core.**experiment**(*data_param=None*)
Decorator that adds logging functionality to experiment methods.

Parameters

- **data_param** (*str*) – Parameter treated by the method as data
- **All other parameters are treated as experimental parameters.** (*input.*) –

Returns A decorator that adds bookkeeping functionality to its argument.

Return type func

decu.core.**figure**(*show=False*, *save=True*)
Create the figure decorator.

Parameters

- **show** (*bool*) – Whether or not the figure should be shown.
- **save** (*bool*) – Whether or not the figure should be saved to disk.

Returns A decorator that adds figure logging functionality to its argument.

Return type func

decu.core.**run_parallel**(*exp*, *params*)
Run an experiment in parallel.

For each element *p* in *params*, call *exp(*p)*. These calls are made in parallel using multiprocessing.

Parameters

- **exp** (*method*) – A @experiment-decorated method.
- **params** (*list*) – Each element is a set of arguments to call *exp* with.

Returns The result of calling *exp(*pi)* over each element of params.

Return type list

exception decu.core.**DecuException**
Bases: Exception

4.2 io.py

Reading and writing functionality for decu.

decu.io.**write**(*result*, *basename*)
Write result to disk.

decu.io.**read**(*infile*)
Read result from disk.

4.3 logging.py

Logging system setup for decu, specially, make multiprocessing and logging play nicely together.

class decu.logging.**DecuLogger**(*start_time*, *project_dir*, *module*)
Bases: object

critical(*msg*)

debug(*msg*)

error(*msg*)

info(*msg*)

log(*level*, *msg*)

warning(*msg*)

4.4 config.py

Custom configuration parsing for decu, based on configparser.

class decu.config.**DecuParser**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section='DEFAULT'*, *interpolation=<object object>*, *converters=<object object>*)

Bases: configparser.ConfigParser

ConfigParser subclass for decu.

Treat every option in the config file as a string.Template object.

subs(*section*, *option*, ***kwargs*)

Perform the named substitutions on the option.

Each option in the decu configuration file is treated as a string.Template object, and thus accepts named variables for substitution. To read a raw option from the configuration file, do decu.config[section][option]. To read a substituted option, do decu.config.subs(section, option, kwargs) or decu.config.subs[section].subs(option, kwargs), where every keyword argument is a name=string pair to be substituted in the option.

Parameters

- **section** (*str*) – the desired section.
- **option** (*str*) – the desired option.
- **kwargs** (*dict*) – every keyword argument is a name=string pair to be substituted in the option template.

Returns the option template with the named strings substituted in.

Return type str

Python Module Index

d

Index