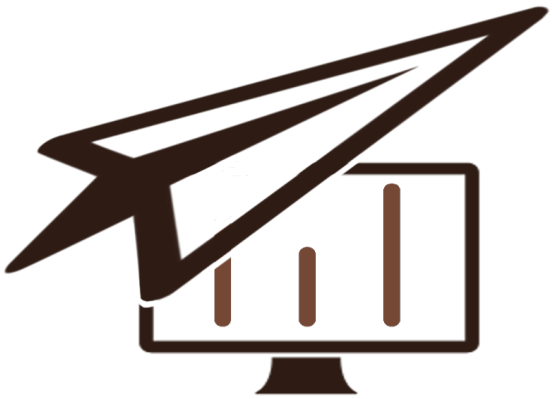
doce

Mathieu Lagrange

Apr 21, 2023

GETTING STARTED



doce is a python package for handling numerical experiments using a design-of-experiment (DOE) approach. It is geared towards research in machine learning and data science but also be useful for other fields.

For a quick introduction to using doce, please refer to the *Tutorial*.

CHAPTER

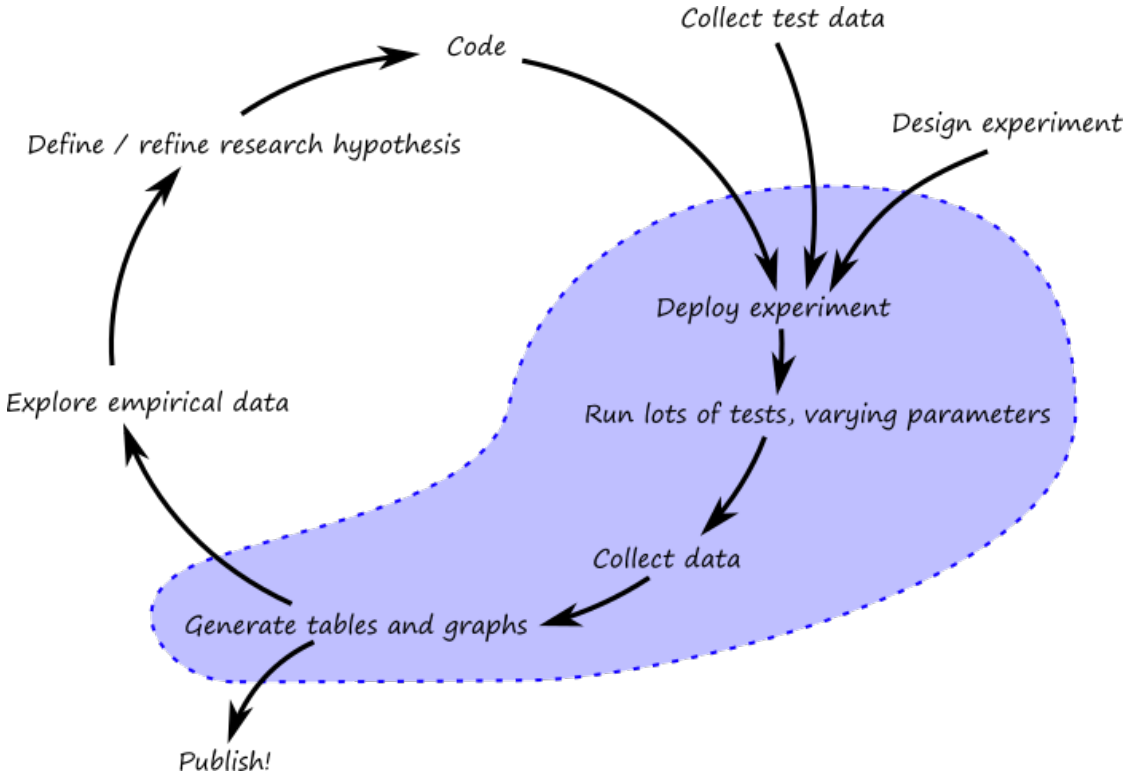
ONE

WHAT FOR ?

Experimentation and testing are a fundamental part of scientific research. They're also extremely time-consuming, tedious, and error-prone unless properly managed. *doce* is here to help you with the management, running and reporting of your experiments, letting you focus more on the creative part of your research work.

What can *doce* can help you to do?

- Set up an experimental protocol once, and re-run it as often as needed with minimum hassle using new data, modified code, or alternative hypotheses.
- Systematically explore many combinations of operational parameters in a system.
- Re-use earlier results to only compute the later stages of a process, thus cutting down the running time.
- Automatically produce analysis tables.
- Ease the production of paper-ready displays.
- Keep experimental code, data and results organized in a standardized way, to ease cooperation with others and allow you to go back painlessly on months-old or years-old work.



1.1 Installation instructions

1.1.1 pypi

The simplest way to install *doce* is through the Python Package Index (PyPI). This will ensure that all required dependencies are fulfilled. This can be achieved by executing the following command:

```
pip install doce
```

or:

```
sudo pip install doce
```

to install system-wide, or:

```
pip install -u doce
```

to install just for your own user.

1.1.2 Source

The latest development version can be installed via pip:

```
pip install git+https://github.com/mathieulagrange/doce
```

1.2 Tutorial

This section covers the fundamentals of developing with *doce*, including a package overview, basic and advanced usage. We will assume basic familiarity with Python and NumPy.

1.2.1 Overview

The *doce* package is structured as collection of submodules that are each responsible for the important parts of managing a computational experiment:

- ***doce.cli***
  Command-line interaction.
- ***doce.experiment***
  Specify every aspects of the experiments from naming, storage location, plan, etc…
- ***doce.plan***
  Generate a number of settings by selecting factors and modalities of a given plan.
- ***doce.setting***
  Manipulate the settings generated by the plan.
- ***doce.metric***
  Manipulate and retrieve the output data.
- ***doce.util***
  Utility functions.

1.2.2 Quickstart

The *doce* package is designed to require very few lines of code around your processing code to handle the task of evaluating its performance with respect to different parametrizations.

Define the experiment

In a .py file, ideally named after the name of your experiment, you have to implement a *set* function that contains the relevant definition of your experiment. The demonstrations discussed in this tutorial are available in the examples directory of the github repository of the project. In this first example, the demo.py is considered.

```
# define the experiment
experiment = doce.Experiment(
  name = 'demo',
  purpose = 'hello world of the doce package',
  author = 'mathieu Lagrange',
  address = 'mathieu.lagrange@ls2n.fr',
)
# set acces paths (here only storage is needed)
experiment.set_path('output', '/tmp/'+experiment.name+'/')
# set some non varying parameters (here the number of cross validation folds)
experiment.n_cross_validation_folds = 10
```

Define the plan

In *doce*, the parametrization of the processing code is called a *setting*. Each setting is a set of *factors*, each factor being uniquely instantiated by a *modality*, chosen among a pre-defined set of modalities.

```
  # set the plan (factor : modalities)
  experiment.add_plan('plan',
    nn_type = ['cnn', 'lstm'],
    n_layers = np.arange(2, 10, 3),
    learning_rate = [0.001, 0.0001],
    dropout = [0, 1]
  )
```

Interact with your experiment

The *doce* package have a convenient way of interacting with experiments, through the command-line. For this to work, you need to add those lines to your python file:

```
# invoke the command line management of the doce package
if __name__ == "__main__":
  doce.cli.main(experiment = experiment,
          func = step
          )
```

Now you can interact with your experiment. For example you can display the plan:

```
$ python demo.py -p
         Factors      0       1  2
0        nn_type    cnn    lstm
1       n_layers      2       5  8
2  learning_rate  0.001  0.0001  0.00001
3        dropout      0       1
```

You can also access to a reference list of each pre-defined argument:

```
$ python demo.py -h
usage: demo.py [-h] [-A [ARCHIVE]] [-C] [-d [DISPLAY]] [-e [EXPORT]] [-H HOST] [-i] [-K␣
↪[KEEP]] [-l]
               [-M [MAIL]] [-p] [-P [PROGRESS]] [-r [RUN]] [-R [REMOVE]] [-s SELECT] [-
↪S] [-u USERDATA]
               [-v] [-V]

optional arguments:
  -h, --help            show this help message and exit
...
```

Control the plan

You can list the different settings generated by the plan:

```
$ python demo.py -l
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=1
nn_type=cnn+n_layers=2+learning_rate=0dot0001+dropout=0
... (36 lines)
```

Most of the time you want to process or retrieve the output data of a *selection* of settings. Doce provides 3 selection formats for expressing that selection :

1. the string format,
2. the dictionary format,
3. the numeric array format.

Suppose you want to select the settings with n_layers=2 and no dropout, you can do that easily with a string formatted selector:

```
python demo.py -l -s n_layers=2+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot0001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=0
nn_type=lstm+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=lstm+n_layers=2+learning_rate=0dot0001+dropout=0
nn_type=lstm+n_layers=2+learning_rate=1edash05+dropout=0
```

Suppose you want to select the settings with nn_type=cnn, n_layers=2, n_layers=8 and no dropout with the string format, the only way is to chain selectors:

```
$ python demo.py -l -s nn_type=cnn+n_layers=2+dropout=0,nn_type=cnn+n_layers=5+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot0001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=0
nn_type=cnn+n_layers=5+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=5+learning_rate=0dot0001+dropout=0
nn_type=cnn+n_layers=5+learning_rate=1edash05+dropout=0
```

This can get tedious when you want to select multiple modalities for multiple factors. For example, suppose you want to select the settings with nn_type=cnn, n_layers=[2, 4] and learning_rate= [0.001, 0.00001], you can do that conveniently with a dictionary formatted selector:

```
$ python demo.py -l -s '{"nn_type"="cnn", "n_layers":[2, 5],"learning_rate":[0.001,0.
↪00001]}'
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=1
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=0
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=1
nn_type=cnn+n_layers=5+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=5+learning_rate=0dot001+dropout=1
nn_type=cnn+n_layers=5+learning_rate=1edash05+dropout=0
nn_type=cnn+n_layers=5+learning_rate=1edash05+dropout=1
```

The ‘’ delimiters are required to avoid interpretetation of the selector by the shell. The ” inside the selector delimiters *must not* be replaced by ‘’ delimiters.

You can perform the same selection with a numeric array formatted selector:

```
$ python demo.py -l -s '[0,[0, 1],[0, 2]]'
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=2+learning_rate=0dot001+dropout=1
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=0
nn_type=cnn+n_layers=2+learning_rate=1edash05+dropout=1
nn_type=cnn+n_layers=5+learning_rate=0dot001+dropout=0
nn_type=cnn+n_layers=5+learning_rate=0dot001+dropout=1
nn_type=cnn+n_layers=5+learning_rate=1edash05+dropout=0
nn_type=cnn+n_layers=5+learning_rate=1edash05+dropout=1
```

As with the string selector, the dict and numeric array types of selector can be chained with a ,.

Define processing code

You must define which code shall be processed for any setting, given the computing environnent defined by the experiment by implementing a step function:

```
def step(setting, experiment):
  # the accuracy  is a function of cnn_type, and use of dropout
  accuracy = (len(setting.nn_type)+setting.dropout+np.random.random_sample(experiment.n_
↪cross_validation_folds))/6
  # duration is a function of cnn_type, and n_layers
  duration = len(setting.nn_type)+setting.n_layers+np.random.randn(experiment.n_cross_
↪validation_folds)
  # storage of outputs (the string between _ and .npy must be the name of the metric␣
```

(continues on next page)

(continued from previous page)

```
↪defined in the set function)
  np.save(experiment.path.output+setting.id()+'_accuracy.npy', accuracy)
  np.save(experiment.path.output+setting.id()+'_duration.npy', duration)
```

In this demo, the processing code simply stores some dummy outputs to the disk.

Perform computation

Now that we have set all this, performing the computation of some settings can simply be done by:

```
$ python demo.py -c -s '{"nn_type":"cnn", "n_layers":[2, 5],"learning_rate":[0.001,0.
↪00001]}'
```

Adding a **-P** to the command line conveniently displays a per setting progress bar.

Removing the **-s** will require the computation of all the settings.

Some settings can fail, which will stop the entire loop. If you want to compute all the non failing settings, you can use the detached computation mode, available with **-D**.

If some settings have failed, a log file is available to provide guidance for debugging your code.

Once fixed, you can be interested in computing only the settings that have failed. For this, you can use the skipping computation mode, available with **-S**. In that mode, for each setting, doce will search for available metrics. If available, the setting is not computed.

Warning: do not consider skipping if some settings have been previously succesfully computed using an outdated version of your code.

Define metrics

Before inspecting the results of our computation, we have to define how the output stored on disc shall be reduced to metrics for interpretation purposes.

To do so, we have to use the `set_metric()` method.

```
# set the metrics
experiment.set_metric(
  name = 'accuracy',
  percent=True,
  higher_the_better= True,
  significance = True,
  precision = 10
  )

experiment.set_metric(
  name = 'acc_std',
  output = 'accuracy',
  func = np.std,
  percent=True
  )

# custom metric function shall input an np.nd_array and output a scalar
def my_metric(data):
```

(continues on next page)

(continued from previous page)

```
  return np.sum(data)

experiment.set_metric(
  name = 'acc_my_metric',
  output = 'accuracy',
  func = my_metric,
  percent=True
  )
```

Display metrics

The reduced version of the metrics can be visualized in the command-line using **-d** :

```
$ python demo.py -d
Displayed data generated from Mon Mar 21 13:59:13 2022 to Mon Mar 21 13:59:13 2022
nn_type: cnn
   n_layers  learning_rate  dropout  accuracyMean%+  accuracyStd%  durationMean*-
0         2        0.00100        0            58.0           5.0            5.63
1         2        0.00100        1            74.0           5.0            5.21
2         2        0.00001        0            56.0           4.0            4.67
3         2        0.00001        1            78.0           3.0            4.81
4         5        0.00100        0            56.0           4.0            8.44
5         5        0.00100        1            76.0           5.0            8.20
6         5        0.00001        0            60.0           6.0            8.59
7         5        0.00001        1            75.0           4.0            7.90
```

Only the metrics available on disc are considered in the table.

You can select the metrics you want to display. To display one metric:

```
$ python demo.py -d 0
Displayed data generated from Mon May 16 15:56:16 2022 to Mon May 16 15:56:16 2022
nn_type: cnn
   n_layers  learning_rate  dropout  accuracyMean%+
0         2        0.00100        0              58
...
```

To display an arbitrary number of metrics, say first and third:

```
$ python demo.py -d '[0, 2]'
Displayed data generated from Mon May 16 15:56:16 2022 to Mon May 16 15:56:16 2022
nn_type: cnn
   n_layers  learning_rate  dropout  accuracyMean%+  durationMean*-
0         2        0.00100        0              58            4.31
```

doce allows you to analyse the impact of a given factor on a given metric. for example, let us study the impact of **n_layers** on **durationMean**:

```
$ python demo.py -d 2:n_layers  -s '{"nn_type":"cnn", "n_layers":[2, 5],"learning_rate
↪":[0.001,0.00001]}'

Displayed data generated from Mon May 16 16:47:38 2022 to Mon May 16 16:47:38 2022
```

(continues on next page)

(continued from previous page)

```
metric: durationMean*- for factor nn_type: cnn  n_layers
   learning_rate  dropout     2     5
0        0.00100        0  5.32  8.14
1        0.00100        1  4.85  7.69
2        0.00001        0  5.43  8.20
3        0.00001        1  5.54  7.98
```

Note that here you have to provide the selector for **doce** to infer the correct organization of the table. This command will fail if some of the needed settings are not available.

Export metrics

The table can exported in various format:

- html
- pdf
- png
- tex
- csv
- xls

To export the table in files called demo, please type : .. code-block:: console

```
$ python demo.py -d -e demo
```

To only generate the html output, please type : .. code-block:: console

```
$ python demo.py -d -e demo.html
```

For visualization purposes, the html output is perhaps the most interesting one, as it shows best values per metrics and statistical analysis :

nn_type: cnn

	n_layers	learning_rate	dropout	accuracyMean%+	accuracyStd%	durationMean*-
0	2	0.001000	0	58	5	5.63
1	2	0.001000	1	74	5	5.21
2	2	0.000010	0	56	4	**4.67**
3	2	0.000010	1	**78**	3	4.81
4	5	0.001000	0	56	4	8.44
5	5	0.001000	1	76	5	8.20
6	5	0.000010	0	60	6	8.59
7	5	0.000010	1	75	4	7.90

The title specifies the factors with unique modality in the selection.

Please note that the page as an auto-reload javascript code snippet that conveniently reloads the page at each new focus.

The mean accuracy is defined as a higher-the-better metric; thus 78 is displayed in bold. the average duration is specified as a lower-the-better metric the 4.67 is displayed in bold. A statistical analysis as been requested (with the *), the several t-tests are operated to check whether the best setting can be assumed to be significantly better than the others. In our example, the other settings with n_layers=2 cannot be assumed to be slower than the most rapid setting.

Mine metrics

Reduced versions of the metrics are convenient to quickly analyse the data. For more refined purposes, such as designing a custom designed plot, one needs to have access to the raw data saved during the processing.

For this example, let us first compute the performance of the cnn and lstm system at a given number of layers and learning with or without dropout:

```
$ python demo.py -s '{"nn_type":["cnn", "lstm"],"n_layers":2,"learning_rate":0.001}' -c
```

Within a python file or a jupyer notebook, we can now retrieve the accuracy data:

```
# your experiment file shall be in the current directory or in the python path
import demo

experiment = demo.set()
selector = {"nn_type":["cnn", "lstm"],"n_layers":2,"learning_rate":0.001}

(data, settings, header) = experiment.get(
  metric = 'accuracy',
  selector = selector,
  path = 'output'
  )
```

The data is a list of np.arrays, the settings is a list of str and the header is a str describing the constant factors. data and settings are of the same size.

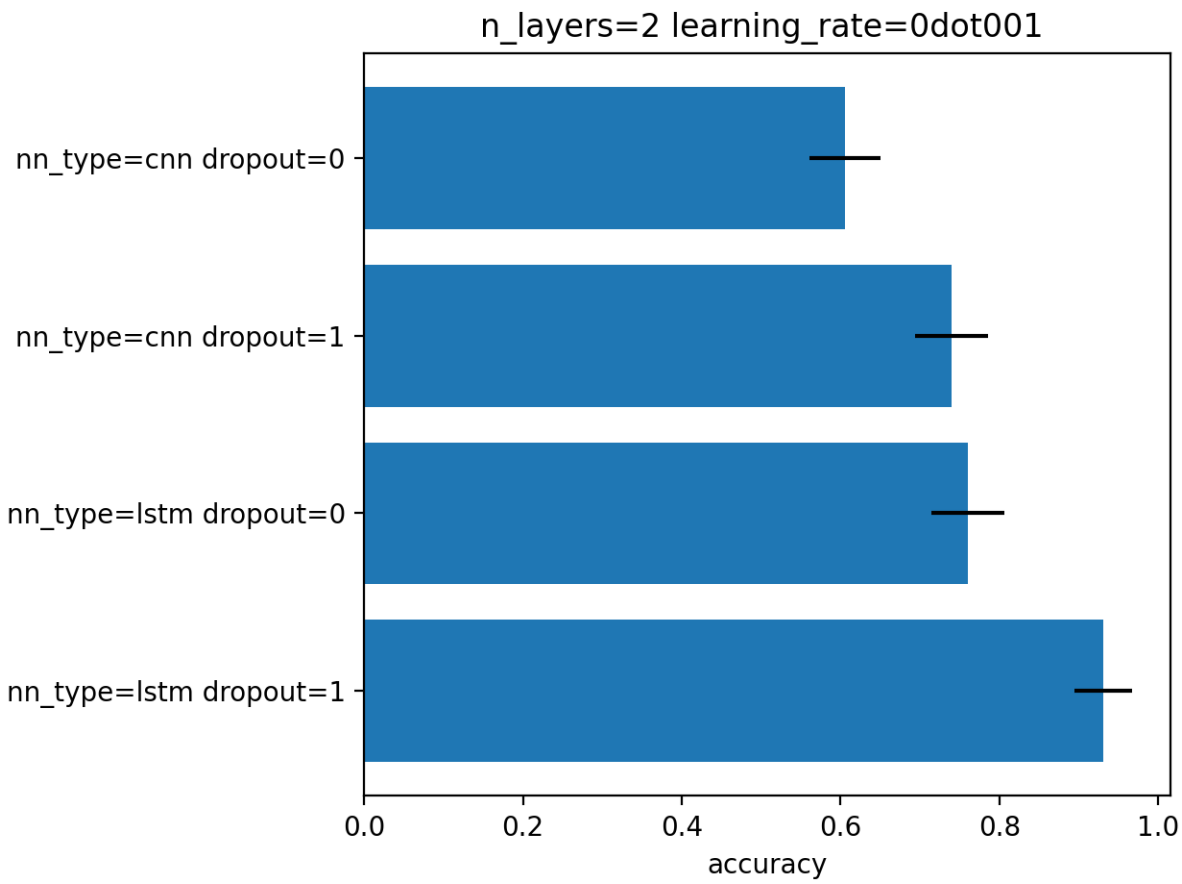
In our example, the data can be conveniently displayed using any horizontal bar plot:

```
import numpy as np
import matplotlib.pyplot as plt

settingIds = np.arange(len(description))

fig, ax = plt.subplots()
ax.barh(settingIds, np.mean(data, axis=1), xerr=np.std(data, axis=1), align='center')
ax.set_yticks(settingIds)
ax.set_yticklabels(settings)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Accuracy')
ax.set_title(header)

fig.tight_layout()
plt.show()
```



1.2.3 Customizing the plan

The definite plan for a given experiment is only known when the experiment is over. It is therefore important to be able to fine tune the plan along with your exploration.

This is not trivial to achieve as it may lead to inconsistencies in stored metric naming conventions if not properly handled.

If you are looking for adding another whole new algorithm or processing step to your experiment, it may be worth considering multiple plans, as described in the dedicated section.

Adding a modality

The addition of a modality is simply done by adding a value to the array of a given factor.

Note that order of modalities matters as it will determine the order in which settings are computed. This is convenient, because you can assume that when requesting the computation of all steps, the output data of step1 will be available to step2, and so on.

Important, this assertion no longer holds if parallelization over settings is selected.

Removing a modality

The removal of a modality is simply done by removing the value to the array of a given factor.

If you want to discard the output data that is no longer accessible, you can do it manually by considering the rm command. Let us assume that we want to remove the modality 0.001 from the factor learning_rate. You can type:

```
$ rm *learning_rate=0dot001*.npy <insert_path>
```

You can also do *before* removing the modality in the array:

```
$ python demo.py -R output -s learning_rate=0dot001
INFORMATION: setting path.archive allows you to move the unwanted files to the archive␣
↪path and not delete them.
List the 24 files ? [Y/n]
/tmp/demo/dropout=0+learning_rate=0dot001+n_layers=8+nn_type=lstm_accuracy.npy
...
/tmp/demo/dropout=0+learning_rate=0dot001+n_layers=8+nn_type=cnn_accuracy.npy
About to remove 24 files from /tmp/demo/
 Proceed ? [Y/n]
```

The selector can be more precise that just one modality.

Adding a factor

Let us say you are considering two classifiers in your experiment: as cnn based and a lstm (code is available in the example directory under file factor_manipulation.py). The plan would be:

```
experiment.addPlan('plan',
  nn_type = ['cnn', 'lstm'],
  # dropout = [0, 1]
)
```

Please note the dropout factor is commented for now. The step function simply saves a .npy file with a 0 value in it. Thus, the output directory contains:

```
$ ls -1 /tmp/factor_manipulation/
nn_type=cnn_accuracy.npy
nn_type=lstm_accuracy.npy
```

And the display command will show:

```
$ python factor_manipulation.py -d
Displayed data generated from Thu Mar 24 10:02:24 2022 to Thu Mar 24 10:02:24 2022

  nn_type  accuracyMean
0     cnn           0.0
1    lstm           0.0
```

Now, let's add the dropout factor by uncommenting its line in the plan:

```
experiment.addPlan('plan',
  nn_type = ['cnn', 'lstm'],
  dropout = [0, 1]
)
```

Now, the problem is that the display command will show nothing:

```
$ python factor_manipulation.py -d
```

Why is that ? Well, now that we have added a new factor, the settings file list is:

```
$ python factor_manipulation.py -f
dropout=0+nn_type=cnn
dropout=1+nn_type=cnn
dropout=0+nn_type=lstm
dropout=1+nn_type=lstm
```

which do not match any of the stored files. In this example, we could simply recompute dropout=0+nn_type=cnn and dropout=0+nn_type=lstm, but in production, that could mean a loss of lengthy computations. The solution to this critical problem is to explicitly state a default value for the factor dropout:

```
experiment.default(plan='plan', factor='dropout', modality=0)
```

Now the settings file list is:

```
$ python factor_manipulation.py -f
nn_type=cnn
dropout=1+nn_type=cnn
nn_type=lstm
dropout=1+nn_type=lstm
```

And the previously computed metrics can now be displayed as before:

```
$ python factor_manipulation.py -d
Displayed data generated from Thu Mar 24 10:02:24 2022 to Thu Mar 24 10:02:24 2022
dropout: 0
  nn_type  accuracyMean
0     cnn           0.0
1    lstm           0.0
```

Removing a factor

Important, this kind of manipulation may lead to output data loss. Be sure to make a backup before attempting to remove a factor.

Let us consider that you have tested whether dropout is useful or not and have decided that dropout is always useful and that you want to remove the dropout factor to avoid clutter in the plan.

Simply removing the factor will lead to the need to redo every computation. It is thus required to perform the following steps:

1. keep only wanted settings (here settings with dropout=0)
2. rename files by removing reference to the dropout setting.

Let us assume that we have computed every settings, the files are:

```
$ python factor_manipulation.py -c
$ ls -1 /tmp/factor_manipulation/
dropout=1+nn_type=cnn_accuracy.npy
dropout=1+nn_type=lstm_accuracy.npy
```

(continues on next page)

(continued from previous page)

```
nn_type=cnn_accuracy.npy
nn_type=lstm_accuracy.npy
```

Keeping only the files of interest is done so:

```
$ python factor_manipulation.py -K output -s dropout=1
INFORMATION: setting path.archive allows you to move the unwanted files to the archive␣
↪path and not delete them.
List the 2 files ? [Y/n]
/tmp/factor_manipulation/nn_type=cnn_accuracy.npy
/tmp/factor_manipulation/nn_type=lstm_accuracy.npy
About to remove 2 files from /tmp/factor_manipulation/
 Proceed ? [Y/n]
```

To rename files by removing reference to the dropout setting.

```
$ rename -n 's/(\+)?dropout=1(\+)?(_)?/$3/' /tmp/factor_manipulation/*npy
Use of uninitialized value $3 in substitution (s///) at (eval 2) line 1.
'/tmp/factor_manipulation/dropout=1+nn_type=cnn_accuracy.npy' would be renamed to '/tmp/
↪factor_manipulation/nn_type=cnn_accuracy.npy'
Use of uninitialized value $3 in substitution (s///) at (eval 2) line 1.
'/tmp/factor_manipulation/dropout=1+nn_type=lstm_accuracy.npy' would be renamed to '/tmp/
↪factor_manipulation/nn_type=lstm_accuracy.npy'
```

Check that the correct files are targeted and remove the -n in the command. Now you can safely remove the dropout factor from the plan.

Managing multiple plans

Most of the time, computational approaches have different needs in terms of parametrization, which add difficulties in managing plans of computations. The doce package handle this by allowing the definition of multiple plans that are then automatically merged is needed. In this first example, the demo_multiple_plan.py is considered.

Assume that we want to compare 3 classifiers : 1. an svm 2. a cnn 3. an lstm

The last two classifiers share the same factors, but the svm have only one factor, called c.

We start by defining the “svm” plan:

```
# set the "svm" plan
experiment.addPlan('svm',
  classifier = ['svm'],
  c = [0.001, 0.0001, 0.00001]
)
```

We then define the “deep” plan:

```
# set the "deep" plan
experiment.addPlan('deep',
  classifier = ['cnn', 'lstm'],
  n_layers = [2, 4, 8],
  dropout = [0, 1]
)
```

Selecting a given plan is done using the selector:

```
$ python demo_multiple_plan.py  -s svm/ -l
Plan svm is selected
classifier=svm+c=0dot001
classifier=svm+c=0dot0001
classifier=svm+c=1edash05
```

Otherwise, the merged plan is considered:

```
$ python demo_multiple_plan.py  -p
Plan svm:
      Factors      0       1      2
0  classifier    svm
1           c  0.001  0.0001  1e-05
Plan deep:
      Factors    0     1  2
0  classifier  cnn  lstm
1    n_layers    2     4  8
2     dropout    0     1
Those plans can be selected using the selector parameter.
Otherwise the merged plan is considered:
      Factors      0      1       2      3
0  classifier    svm    cnn    lstm
1           c  *0.0*  0.001  0.0001  1e-05
2    n_layers    *0*      2       4      8
3     dropout    *0*      1
```

Computation can be done using the specified plans:

```
$ python demo_multiple_plan.py  -s svm/ -c
Plan svm is selected
$ python demo_multiple_plan.py  -s deep/ -c
Plan deep is selected
```

Display of metric is conveniently done using the merged plan:

```
$ python demo_multiple_plan.py  -d
Displayed data generated from Mon Mar 21 17:22:32 2022 to Mon Mar 21 17:26:22␣
↪2022

  classifier     c  n_layers  dropout  accuracyMean%
0        svm  1.00         0        0            8.0
1        svm  0.10         0        0            1.0
2        svm  0.01         0        0            0.0
3        cnn  0.00         2        1           76.0
4        cnn  0.00         4        1           74.0
5        cnn  0.00         8        1           77.0
6       lstm  0.00         2        1           94.0
7       lstm  0.00         4        1           91.0
8       lstm  0.00         8        1           91.0
```

1.2.4 Advanced usage

Tagging computations

During development, it is sometimes useful to differentiate between several runs of the experiment. For example, you might want to try out a new tweak or play around with some parameters that you do not want to add to the plan.

You can do so by tagging. The tag will add a level of hierarchy in your output paths. Let us assume that you have python file demo.py that defines a storage path named output pointing to /tmp/experiment/. When running python demo.py –tag my_tag, the storage path will now be /tmp/experiment/my_tag.

This gives you the freedom to switch easily to compare relative performance. For replication purposes, this tag can conveniently be defined as an id of your prefered code versioning system.

If you want tag outputs to become the default outputs, you simply have to move file from the tag directory to the root directory. In this example, mv /tmp/experiment/my_tag/* /tmp/experiment.

Storage within an hdf5 file

Remote computing

1.3 Cli

Handle interaction with the doce module using the command line interface.

doce.cli.**main**(*experiment=None*, *func=None*, *display_func=None*)

This method shall be called from the main script of the experiment to control the experiment using the command line.

This method provides a front-end for running a doce experiment. It should be called from the main script of the experiment. The main script must define the **experiment** object that will be called before processing and a **func** function that will be run for each setting.

Examples

Assuming that the file experiment_run.py contains:

```
>>> import doce
>>> if __name__ == "__main__":
...   doce.experiment.run()
>>> def set(experiment, args=None):
...   experiment._plan.factor1=[1, 3]
...   experiment._plan.factor2=[2, 4]
...   return experiment
>>> def step(setting, experiment):
...   print(setting.identifier())
```

Executing this file with the –run option gives:

$ python experiment_run.py -r
factor1_1_factor2_2 factor1_1_factor2_4 factor1_3_factor2_2 factor1_3_factor2_4

Executing this file with the –help option gives:

$ python experiment_run.py -h

1.4 Experiment

Handle information of an experiment of the doce module.

class doce.experiment.**Experiment**(***description*)

Stores high level information about the experiment and tools to control the processing and storage of data.

The experiment class displays high level information about the experiment such as its name, description, author, author's email address, and run identification.

Information about storage of data is specified using the experiment.path name_space. It also stores one or several Plan objects and a Metric object to respectively specify the experimental plans and the metrics considered in the experiment.

See also:

doce.Plan, *doce.metric.Metric*

Examples

```
>>> import doce
>>> e=doce.Experiment()
>>> e.name='my_experiment'
>>> e.author='John Doe'
>>> e.address='john.doe@no-log.org'
>>> e.path.processing='/tmp'
>>> print(e)
  name: my_experiment
  description
  author: John Doe
  address: john.doe@no-log.org
  version: 0.1
  status:
    run_id: ...
    verbose: 0
  selector: []
  parameter
  metric
  path:
    code_raw: ...
    code: ...
    archive_raw:
    archive:
    export_raw: export
    export: export
    processing_raw: /tmp
    processing: /tmp
  host: []
```

Each level can be complemented with new members to store specific information:

```
>>> e.specific_info = 'stuff'
>>> import types
```

(continues on next page)

(continued from previous page)

```
>>> e.my_data = types.SimpleNamespace()
>>> e.my_data.info1= 1
>>> e.my_data.info2= 2
>>> print(e)
  name: my_experiment
  description
  author: John Doe
  address: john.doe@no-log.org
  version: 0.1
  status:
    run_id: ...
    verbose: 0
  selector: []
  parameter
  metric
  path:
    code_raw: ...
    code: ...
    archive_raw:
    archive:
    export_raw: export
    export: export
    processing_raw: /tmp
    processing: /tmp
  host: []
  specific_info: stuff
  my_data:
    info1: 1
    info2: 2
```

Methods

add_setting_group(file_id, setting[, ...])	adds a group to the root of a valid py_tables Object in order to store the metrics corresponding to the specified setting.
clean_data_sink(path[, selector, reverse, ...])	Perform a cleaning of a data sink (directory or h5 file).
get_output([output, selector, path, tag, plan])	Get the output vector from an .npy or a group of a .h5 file.
perform(selector[, function, nb_jobs, ...])	Operate the function with parameters on the *settings* set generated using *selector*.
send_mail([title, body])	Send an email to the email address given in experiment.address.
set_path(name, path[, force])	Create directories whose path described in experiment.path are not reachable.

add_plan	
default	
get_current_plan	
plans	
select	
set_metric	
skip_setting	

__str__(*style='str'*)

Provide a textual description of the experiment

List all members of the class and theirs values

Parameters

style
[str] If 'str', return the description as a string.

If 'html', return the description with an html format.

Returns

description
[str] If style == 'str' : a carriage return separated enumeration of the members of the class experiment.

If style == 'html' : an html version of the description

Examples

```
>>> import doce
>>> print(doce.Experiment())
name
description
author: no name
address: noname@noorg.org
version: 0.1
status:
  run_id: ...
  verbose: 0
selector: []
parameter
metric
path:
  code_raw: ...
  code: ...
  archive_raw:
  archive:
  export_raw: export
  export: export
host: []
```

```
>>> import doce
>>> doce.Experiment().__str__(style='html')
```

(continues on next page)

(continued from previous page)

```
    '<div>name</div><div>description</div><div>author: no name</div><div>
↪address: noname@noorg.org</div><div>version: 0.1</div><div>status:</div><div>␣
↪ run_id: ...</div><div>  verbose: 0</div><div>selector: []</div><div>parameter
↪</div><div>metric</div><div>path:</div><div>  code_raw: ...</div><div>  code:␣
↪...</div><div>  archive_raw: </div><div>  archive: </div><div>  export_raw:␣
↪export</div><div>  export: export</div><div>host: []</div><div></div>'
```

add_setting_group(*file_id*, *setting*, *output_dimension=None*, *setting_encoding=None*)

adds a group to the root of a valid py_tables Object in order to store the metrics corresponding to the specified setting.

adds a group to the root of a valid py_tables Object in order to store the metrics corresponding to the specified setting. The encoding of the setting is used to set the name of the group. For each metric, a Floating point Pytable Array is created. For any metric, if no dimension is provided in the output_dimension dict, an expandable array is instantiated. If a dimension is available, a static size array is instantiated.

Parameters

file_id: py_tables file Object
a valid py_tables file Object, leading to an .h5 file opened with writing permission.
setting: :class:`doce.Plan`
an instantiated Factor object describing a setting.
output_dimension: dict
for metrics for which the dimensionality of the storage vector is known,
each key of the dict is a valid metric name and each corresponding value
is the size of the storage vector.
setting_encoding
[dict]

Encoding of the setting. See doce.Plan.id for references.

Returns

setting_group: a Pytables Group
where metrics corresponding to the specified setting are stored.

Examples

```
>>> import doce
>>> import numpy as np
>>> import tables as tb
```

```
>>> experiment = doce.experiment.Experiment()
>>> experiment.name = 'example'
>>> experiment.set_path('output', '/tmp/'+experiment.name+'.h5')
>>> experiment.add_plan('plan', f1 = [1, 2], f2 = [1, 2, 3])
>>> experiment.set_metric(name = 'm1_mean', output = 'm1', func = np.mean)
>>> experiment.set_metric(name = 'm1_std', output = 'm1', func = np.std)
>>> experiment.set_metric(name = 'm2_min', output = 'm2', func = np.min)
>>> experiment.set_metric(name = 'm2_argmin', output = 'm2', func = np.argmin)
```

```
>>> def process(setting, experiment):
...  h5 = tb.open_file(experiment.path.output, mode='a')
```

(continues on next page)

(continued from previous page)

```
...   sg = experiment.add_setting_group(h5, setting, output_dimension = {'m1
↪':100})
...   sg.m1[:] = setting.f1+setting.f2+np.random.randn(100)
...   sg.m2.append(setting.f1*setting.f2*np.random.randn(100))
...   h5.close()
>>> nb_failed = experiment.perform([], process, progress='')
```

```
>>> h5 = tb.open_file(experiment.path.output, mode='r')
>>> print(h5)
/tmp/example.h5 (File) ''
Last modif.: '...'
Object Tree:
/ (RootGroup) ''
/f1=1+f2=1 (Group) 'f1=1+f2=1'
/f1=1+f2=1/m1 (Array(100,)) 'm1'
/f1=1+f2=1/m2 (EArray(100,)) 'm2'
/f1=1+f2=2 (Group) 'f1=1+f2=2'
/f1=1+f2=2/m1 (Array(100,)) 'm1'
/f1=1+f2=2/m2 (EArray(100,)) 'm2'
/f1=1+f2=3 (Group) 'f1=1+f2=3'
/f1=1+f2=3/m1 (Array(100,)) 'm1'
/f1=1+f2=3/m2 (EArray(100,)) 'm2'
/f1=2+f2=1 (Group) 'f1=2+f2=1'
/f1=2+f2=1/m1 (Array(100,)) 'm1'
/f1=2+f2=1/m2 (EArray(100,)) 'm2'
/f1=2+f2=2 (Group) 'f1=2+f2=2'
/f1=2+f2=2/m1 (Array(100,)) 'm1'
/f1=2+f2=2/m2 (EArray(100,)) 'm2'
/f1=2+f2=3 (Group) 'f1=2+f2=3'
/f1=2+f2=3/m1 (Array(100,)) 'm1'
/f1=2+f2=3/m2 (EArray(100,)) 'm2'
```

```
>>> h5.close()
```

clean_data_sink(*path*, *selector=None*, *reverse=False*, *force=False*, *keep=False*, *wildcard='*'*, *setting_encoding=None*, *archive_path=None*, *verbose=0*)

Perform a cleaning of a data sink (directory or h5 file).

This method is essentially a wrapper to `doce._plan.clean_data_sink()`.

Parameters

path
[str] If has a / or \, a valid path to a directory or .h5 file.

If has no / or \, a member of the name_space self.path.

selector
[a list of literals or a list of lists of literals (optional)] *selector* used to specify the *settings* set

reverse
[bool (optional)] If False, remove any entry corresponding to the setting set (default).

If True, remove all entries except the ones corresponding to the setting set.

force: bool (optional)
If False, prompt the user before modifying the data sink (default).

If True, do not prompt the user before modifying the data sink.

wildcard
[str (optional)] end of the wildcard used to select the entries to remove or to keep (default: '*').

setting_encoding
[dict (optional)] format of the identifier describing the *setting*. Please refer to `doce.Plan.identifier()` for further information.

archive_path
[str (optional)] If not None, specify an existing directory where the specified data will be moved.

If None, the path doce.Experiment._archive_path is used (default).

See also:

doce._plan.clean_data_sink, doce.Plan.id

Examples

```
>>> import doce
>>> import numpy as np
>>> import os
>>> e=doce.Experiment()
>>> e.set_path('output', '/tmp/test', force=True)
>>> e.add_plan('plan', factor1=[1, 3], factor2=[2, 4])
>>> def my_function(setting, experiment):
...   np.save(f'{experiment.path.output}{setting.identifier()}_sum.npy',␣
↪setting.factor1+setting.factor2)
...   np.save(f'{experiment.path.output}{setting.identifier()}_mult.npy',␣
↪setting.factor1*setting.factor2)
>>> nb_failed = e.perform([], my_function, progress='')
>>> os.listdir(e.path.output)
['factor1=1+factor2=4_mult.npy', 'factor1=1+factor2=4_sum.npy',
↪'factor1=3+factor2=4_sum.npy', 'factor1=1+factor2=2_mult.npy',
↪'factor1=1+factor2=2_sum.npy', 'factor1=3+factor2=2_mult.npy',
↪'factor1=3+factor2=4_mult.npy', 'factor1=3+factor2=2_sum.npy']
```

```
>>> e.clean_data_sink('output', [0], force=True)
>>> os.listdir(e.path.output)
['factor1=3+factor2=4_sum.npy', 'factor1=3+factor2=2_mult.npy',
↪'factor1=3+factor2=4_mult.npy', 'factor1=3+factor2=2_sum.npy']
```

```
>>> e.clean_data_sink('output', [1, 1], force=True, reverse=True, wildcard=
↪'*mult*')
>>> os.listdir(e.path.output)
['factor1=3+factor2=4_sum.npy', 'factor1=3+factor2=4_mult.npy',
↪'factor1=3+factor2=2_sum.npy']
```

Here, we remove all the files that match the wildcard *mult* in the directory /tmp/test that do not correspond to the settings that have the first factor set to the second modality and the second factor set to the second modality.

```
>>> import doce
>>> import tables as tb
>>> e=doce.Experiment()
>>> e.set_path('output', '/tmp/test.h5')
>>> e.add_plan('plan', factor1=[1, 3], factor2=[2, 4])
>>> e.set_metric(name = 'sum')
>>> e.set_metric(name = 'mult')
>>> def my_function(setting, experiment):
...   h5 = tb.open_file(experiment.path.output, mode='a')
...   sg = experiment.add_setting_group(
...     h5, setting,
...     output_dimension={'sum': 1, 'mult': 1})
...   sg.sum[0] = setting.factor1+setting.factor2
...   sg.mult[0] = setting.factor1*setting.factor2
...   h5.close()
>>> nb_failed = e.perform([], my_function, progress='')
>>> h5 = tb.open_file(e.path.output, mode='r')
>>> print(h5)
/tmp/test.h5 (File) ''
Last modif.: '...'
Object Tree:
/ (RootGroup) ''
/factor1=1+factor2=2 (Group) 'factor1=1+factor2=2'
/factor1=1+factor2=2/mult (Array(1,)) 'mult'
/factor1=1+factor2=2/sum (Array(1,)) 'sum'
/factor1=1+factor2=4 (Group) 'factor1=1+factor2=4'
/factor1=1+factor2=4/mult (Array(1,)) 'mult'
/factor1=1+factor2=4/sum (Array(1,)) 'sum'
/factor1=3+factor2=2 (Group) 'factor1=3+factor2=2'
/factor1=3+factor2=2/mult (Array(1,)) 'mult'
/factor1=3+factor2=2/sum (Array(1,)) 'sum'
/factor1=3+factor2=4 (Group) 'factor1=3+factor2=4'
/factor1=3+factor2=4/mult (Array(1,)) 'mult'
/factor1=3+factor2=4/sum (Array(1,)) 'sum'
>>> h5.close()
```

```
>>> e.clean_data_sink('output', [0], force=True)
>>> h5 = tb.open_file(e.path.output, mode='r')
>>> print(h5)
/tmp/test.h5 (File) ''
Last modif.: '...'
Object Tree:
/ (RootGroup) ''
/factor1=3+factor2=2 (Group) 'factor1=3+factor2=2'
/factor1=3+factor2=2/mult (Array(1,)) 'mult'
/factor1=3+factor2=2/sum (Array(1,)) 'sum'
/factor1=3+factor2=4 (Group) 'factor1=3+factor2=4'
/factor1=3+factor2=4/mult (Array(1,)) 'mult'
/factor1=3+factor2=4/sum (Array(1,)) 'sum'
```

(continues on next page)

(continued from previous page)

```
>>> h5.close()
```

```
>>> e.clean_data_sink('output', [1, 1], force=True, reverse=True, wildcard=
↪'*mult*')
>>> h5 = tb.open_file(e.path.output, mode='r')
>>> print(h5)
/tmp/test.h5 (File) ''
Last modif.: '...'
Object Tree:
/ (RootGroup) ''
/factor1=3+factor2=4 (Group) 'factor1=3+factor2=4'
/factor1=3+factor2=4/mult (Array(1,)) 'mult'
/factor1=3+factor2=4/sum (Array(1,)) 'sum'
>>> h5.close()
```

Here, the same operations are conducted on a h5 file.

get_output(*output=''*, *selector=None*, *path=''*, *tag=''*, *plan=None*)

Get the output vector from an .npy or a group of a .h5 file.

Get the output vector as a numpy array from an .npy or a group of a .h5 file.

Parameters

output: str
The name of the output.

selector: list
Settings selector.

path: str
Name of path as defined in the experiment, or a valid path to a directory in the case of .npy storage, or a valid path to an .h5 file in the case of hdf5 storage.

plan: str
Name of plan to be considered.

Returns

setting_metric: list of np.Array
stores for each valid setting an np.Array with the values of the metric selected.

setting_description: list of list of str
stores for each valid setting, a compact description of the modalities of each factors. The factors with the same modality accross all the set of settings is stored in constant_setting_description.

constant_setting_description: str
compact description of the factors with the same modality accross all the set of settings.

Examples

```
>>> import doce
>>> import numpy as np
>>> import pandas as pd
```

```
>>> experiment = doce.experiment.Experiment()
>>> experiment.name = 'example'
>>> experiment.set_path('output', '/tmp/{experiment.name}/', force=True)
>>> experiment.add_plan('plan', f1 = [1, 2], f2 = [1, 2, 3])
>>> experiment.set_metric(name = 'm1_mean', output = 'm1', func = np.mean)
>>> experiment.set_metric(name = 'm1_std', output = 'm1', func = np.std)
>>> experiment.set_metric(name = 'm2_min', output = 'm2', func = np.min)
>>> experiment.set_metric(name = 'm2_argmin', output = 'm2', func = np.argmin)
```

```
>>> def process(setting, experiment):
...  output1 = setting.f1+setting.f2+np.random.randn(100)
...  output2 = setting.f1*setting.f2*np.random.randn(100)
...  np.save(f'{experiment.path.output+setting.identifier()}_m1.npy', output1)
...  np.save(f'{experiment.path.output+setting.identifier()}_m2.npy', output2)
>>> nb_failed = experiment.perform([], process, progress='')
```

```
>>> (setting_output,
...  setting_description,
...  constant_setting_description
... ) = experiment.get_output(output = 'm1', selector = [1], path='output')
>>> print(constant_setting_description)
f1=2
>>> print(setting_description)
['f2=1', 'f2=2', 'f2=3']
>>> print(len(setting_output))
3
>>> print(setting_output[0].shape)
(100,)
```

perform(*selector*, *function=None*, **parameters*, *nb_jobs=1*, *progress='d'*, *log_file_name=''*, *mail_interval=0*, *tag=''*)

Operate the function with parameters on the *settings* set generated using *selector*.

Operate a given function on the setting set generated using selector. The setting set can be browsed in parallel by setting nb_jobs>1. If log_file_name is not empty, a faulty setting do not stop the execution, the error is stored and another setting is executed. If progress is set to True, a graphical display of the progress through the setting set is displayed.

This function is essentially a wrapper to the function `doce.Plan.do()`.

Parameters

selector
[a list of literals or a list of lists of literals] *selector* used to specify the *settings* set

function
[function(`Plan`, `Experiment`, *parameters) (optional)] A function that operates on a given setting within the experiment environnment with optional parameters.

If None, a description of the given setting is shown.

***parameters**
[any type (optional)] parameters to be given to the function.

nb_jobs
[int > 0 (optional)] number of jobs.

If nb_jobs = 1, the setting set is browsed sequentially in a depth first traversal of the settings tree (default).

If nb_jobs > 1, the settings set is browsed randomly, and settings are distributed over the different processes.

progress
[str (optional)] display progress of scheduling the setting set.

If str has an m, show the selector of the current setting. If str has an d, show a textual description of the current setting (default).

log_file_name
[str (optional)] path to a file where potential errors will be logged.

If empty, the execution is stopped on the first faulty setting (default).

If not empty, the execution is not stopped on a faulty setting, and the error is logged in the log_file_name file.

mail_interval
[float (optional)] interval for sending email about the status of the run.

If 0, no email is sent (default).

It >0, an email is sent as soon as an setting is done and the difference between the current time and the time the last mail was sent is larger than mail_interval.

tag
[string (optional)] specify a tag to be added to the output names

See also:

`doce.Plan.do`

Examples

```
>>> import time
>>> import random
>>> import doce
```

```
>>> e=doce.Experiment()
>>> e.add_plan('plan', factor1=[1, 3], factor2=[2, 5])
```

```
>>> # this function displays the sum of the two modalities of the current␣
↪setting
>>> def my_function(setting, experiment):
...  print(f'{setting.factor1}+{setting.factor2}={setting.factor1+setting.
↪factor2}')
```

```
>>> # sequential execution of settings
>>> nb_failed = e.perform([], my_function, nb_jobs=1, progress='')
1+2=3
1+5=6
3+2=5
3+5=8
>>> # arbitrary order execution of settings due to the parallelization
>>> nb_failed = e.perform([], my_function, nb_jobs=3, progress='')
3+2=5
1+5=6
1+2=3
3+5=8
```

send_mail(*title=''*, *body=''*)

Send an email to the email address given in experiment.address.

Send an email to the experiment.address email address using the smtp service from gmail. For privacy, please consider using a dedicated gmail account by setting experiment._gmail_id and experiment._gmail_app_password. For this, you will need to create a gmail account, set two-step validation and allow connection with app password.

See https://support.google.com/accounts/answer/185833?hl=en for reference.

Parameters

title
[str] the title of the email in plain text format

body
[str] the body of the email in html format

Examples

```
>>> import doce
>>> e=doce.Experiment()
>>> e.address = 'john.doe@no-log.org'
>>> e.send_mail('hello', '<div> good day </div>')
Sent message entitled: [doce]  id ... hello ...
```

set_path(*name*, *path*, *force=False*)

Create directories whose path described in experiment.path are not reachable.

For each path set in experiment.path, create the directory if not reachable. The user may be prompted before creation.

Parameters

force
[bool] If True, do not prompt the user before creating the missing directories.

If False, prompt the user before creation of each missing directory (default).

Examples

```
>>> import doce
>>> import os
>>> e=doce.Experiment()
>>> e.name = 'experiment'
>>> e.set_path('processing', f'/tmp/{e.name}/processing', force=True)
>>> e.set_path('output', f'/tmp/{e.name}/output', force=True)
>>> os.listdir(f'/tmp/{e.name}')
['processing', 'output']
```

class doce.experiment.**Path**

handle storage of path to disk

doce.experiment.**get_from_path**(*metric*, *settings=None*, *path=''*, *tag=''*, *setting_encoding=None*, *verbose=False*)

Get the metric vector from an .npy or a group of a .h5 file.

Get the metric vector as a numpy array from an .npy or a group of a .h5 file.

Parameters

metric: str
The name of the metric. Must be a member of the doce.metric.Metric object.

settings: doce.Plan
Iterable settings.

path: str
In the case of .npy storage, a valid path to the main directory. In the case of .h5 storage, a valid path to an .h5 file.

setting_encoding
[dict] Encoding of the setting. See doce.Plan.id for references.

verbose
[bool] In the case of .npy metric storage, if verbose is set to True, print the file_name seeked for the metric.

In the case of .h5 metric storage, if verbose is set to True, print the group seeked for the metric.

Returns

setting_metric: list of np.Array
stores for each valid setting an np.Array with the values of the metric selected.

setting_description: list of list of str
stores for each valid setting, a compact description of the modalities of each factors. The factors with the same modality accross all the set of settings is stored in constant_setting_description.

constant_setting_description: str
compact description of the factors with the same modality accross all the set of settings.

Examples

```
>>> import doce
>>> import numpy as np
>>> import pandas as pd
```

```
>>> experiment = doce.experiment.Experiment()
>>> experiment.name = 'example'
>>> experiment.set_path('output', f'/tmp/{experiment.name}/', force=True)
>>> experiment.add_plan('plan', f1 = [1, 2], f2 = [1, 2, 3])
>>> experiment.set_metric(name = 'm1_mean', output = 'm1', func = np.mean)
>>> experiment.set_metric(name = 'm1_std', output = 'm1', func = np.std)
>>> experiment.set_metric(name = 'm2_min', output = 'm2', func = np.min)
>>> experiment.set_metric(name = 'm2_argmin', output = 'm2', func = np.argmin)
>>> def process(setting, experiment):
...  metric1 = setting.f1+setting.f2+np.random.randn(100)
...  metric2 = setting.f1*setting.f2*np.random.randn(100)
...  np.save(f'{experiment.path.output}{setting.identifier()}_m1.npy', metric1)
...  np.save(f'{experiment.path.output}{setting.identifier()}_m2.npy', metric2)
>>> nb_failed = experiment.perform([], process, progress='')
```

```
>>> (setting_metric,
...  setting_description,
...  constant_setting_description) = get_from_path(
...      'm1',
...      experiment._plan.select([1]),
...      experiment.path.output)
>>> print(constant_setting_description)
f1=2
>>> print(setting_description)
['f2=1', 'f2=2', 'f2=3']
>>> print(len(setting_metric))
3
>>> print(setting_metric[0].shape)
(100,)
```

1.5 Plan

Handle storage and processing of the plan of the doce module.

class doce.plan.**Plan**(*name*, ***factors*)

stores the different factors of the doce experiment.

This class stores the different factors of the doce experiments. For each factor, the set of different modalities can be expressed as a list or a numpy array.

To browse the setting set defined by the Plan object, one must iterate over the Plan object.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan('')
>>> p.factor1=[1, 3]
>>> p.factor2=[2, 4]
```

```
>>> print(p)
  0  factor1: [1 3]
  1  factor2: [2 4]
```

```
>>> for setting in p:
...   print(setting)
factor1=1+factor2=2
factor1=1+factor2=4
factor1=3+factor2=2
factor1=3+factor2=4
```

Methods

as_panda_frame()	returns a panda frame that describes the Plan object.
clean_data_sink(path[, reverse, force, ...])	clean a data sink by considering the settings set.
clean_h5(path[, reverse, force, keep, ...])	clean a h5 data sink by considering the settings set.
default(factor, modality)	set the default modality for the specified factor.
factors()	returns the names of the factors.
nb_modalities(factor)	returns the number of *modalities* for a given *factor*.
perform(function, experiment, *parameters[, ...])	iterate over the setting set and run the function given as parameter.
select([selector, volatile, prune])	set the selector.

check	
check_length	
constant_factors	
copy	
expand_selector	
get_name	
merge	
order_factor	

as_panda_frame()

returns a panda frame that describes the Plan object.

Returns a panda frame describing the Plan object.

For ease of definition of a selector to select some settings, the columns and the rows of the panda frame are numbered.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan('')
>>> p.one = ['a', 'b']
>>> p.two = list(range(10))
```

```
>>> print(p)
  0  one: ['a' 'b']
  1  two: [0 1 2 3 4 5 6 7 8 9]
>>> print(p.as_panda_frame())
  Factors  0  1  2  3  4  5  6  7  8  9
0    one  a  b
1    two  0  1  2  3  4  5  6  7  8  9
```

clean_data_sink(*path*, *reverse=False*, *force=False*, *keep=False*, *wildcard='*'*, *setting_encoding=None*, *archive_path=''*, *verbose=0*)

clean a data sink by considering the settings set.

This method is more conveniently used by

considering the method :meth:`doce.experiment._experiment.clean_data_sink, please see its documentation for usage.

clean_h5(*path*, *reverse=False*, *force=True*, *keep=False*, *setting_encoding=None*, *archive_path=''*, *verbose=0*)

clean a h5 data sink by considering the settings set.

This method is more conveniently used by considering

the method :meth:`doce.experiment._experiment.clean_data_sink, please see its documentation for usage.

default(*factor*, *modality*)

set the default modality for the specified factor.

Set the default modality for the specified factor.

Parameters

factor: str
the name of the factor

modality: int or str
the modality value

See also:

doce.Plan.id

Examples

```
>>> import doce
```

p = doce.Plan(‘’)

p.f1 = [‘a’, ‘b’] p.f2 = [1, 2, 3]

print(f) for setting in p.select():

print(setting.identifier())

p.default(‘f2’, 2)

for setting in p:
print(setting.identifier())

p.f2 = [0, 1, 2, 3] print(f)

p.default(‘f2’, 2)

for setting in p:
print(setting.identifier())

factors()

returns the names of the factors.

Returns the names of the factors as a list of strings.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan('')
>>> p.f1=['a', 'b']
>>> p.f2=[1, 2]
>>> p.f3=[0, 1]
```

```
>>> print(p.factors())
['f1', 'f2', 'f3']
```

nb_modalities(*factor*)

returns the number of *modalities* for a given *factor*.

Returns the number of *modalities*

for a given *factor* as an integer value.

Parameters

factor: int or str
if int, considered as the index inside an array of the factors sorted by order of definition.

If str, the name of the factor.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan('')
>>> p.one = ['a', 'b']
>>> p.two = list(range(10))
```

```
>>> print(p.nb_modalities('one'))
2
>>> print(p.nb_modalities(1))
10
```

perform(*function*, *experiment*, **parameters*, *nb_jobs=1*, *progress='d'*, *log_file_name=''*, *mail_interval=0*)

iterate over the setting set and run the function given as parameter.

This function is wrapped by `doce.experiment.Experiment.do()`, which should be more convenient to use. Please refer to this method for usage.

Parameters

function
[function(`Plan`, *Experiment*, *parameters)] operates on a given setting within the experiment environnment with optional parameters.

experiment:
an *Experiment* object

***parameters**
[any type (optional)] parameters to be given to the function.

nb_jobs
[int > 0 (optional)] number of jobs.

If nb_jobs = 1, the setting set is browsed sequentially in a depth first traversal of the settings tree (default).

If nb_jobs > 1, the settings set is browsed randomly, and settings are distributed over the different processes.

progress
[str (optional)] display progress of scheduling the setting set.

If str has an m, show the selector of the current setting. If str has an d, show a textual description of the current setting (default).

log_file_name
[str (optional)] path to a file where potential errors will be logged.

If empty, the execution is stopped on the first faulty setting (default).

If not empty, the execution is not stopped on a faulty setting, and the error is logged in the log_file_name file.

See also:

doce.experiment.Experiment.perform

select(*selector=None*, *volatile=False*, *prune=True*)

set the selector.

This method sets the internal selector to the selector given as parameter.

Once set, iteration over the setting set is limited to the settings that can be reached according to the definition of the selector.

Parameters

selector: list of list of int or list of int or list of dict
a :term:`selector

volatile: bool
if True, the selector is disabled after a complete iteration over the setting set.

If False, the selector is saved for further iterations.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan()
>>> p.f1=['a', 'b', 'c']
>>> p.f2=[1, 2, 3]
```

```
>>> # doce allows two ways of defining the selector. The first one is dict␣
↪based:
>>> for setting in p.select([{'f1':'b', 'f2':[1, 2]}, {'f1':'c', 'f2':[3]}]):
...  print(setting)
f1=b+f2=1
f1=b+f2=2
f1=c+f2=3
```

```
>>> # The second one is list based. In this example, we select the settings with
>>> # the second modality of the first factor, and with the first modality of␣
↪the second factor
>>> for setting in p.select([1, 0]):
...  print(setting)
f1=b+f2=1
>>> # select the settings with all the modalities of the first factor,
>>> # and the second modality of the second factor
>>> for setting in p.select([-1, 1]):
...  print(setting)
f1=a+f2=2
f1=b+f2=2
f1=c+f2=2
>>> # the selection of all the modalities of the remaining factors can be␣
↪conveniently expressed
>>> for setting in p.select([1]):
...  print(setting)
f1=b+f2=1
f1=b+f2=2
f1=b+f2=3
```

(continues on next page)

(continued from previous page)

```
>>> # select the settings using 2 selector, where the first selects the settings
>>> # with the first modalityof the first factor and with the second modality
>>> # of the second factor, and the second selector selects the settings
>>> # with the second modality of the first factor,
>>> # and with the third modality of the second factor
>>> for setting in p.select([[0, 1], [1, 2]]):
...  print(setting)
f1=a+f2=2
f1=b+f2=3
>>> # the latter expression may be interpreted as the selection of the settings␣
↪with
>>> # the first and second modalities of the first factor and with second and
>>> # third modality of the second factor. In that case, one needs to add a -1
>>> # at the end of the selector (even if by doing so the length of the selector
>>> # is larger than the number of factors)
>>> for setting in p.select([[0, 1], [1, 2], -1]):
...  print(setting)
f1=a+f2=2
f1=a+f2=3
f1=b+f2=2
f1=b+f2=3
>>> # if volatile is set to False (default) when the selector is set
>>> # and the setting set iterated, the setting set stays ready for another␣
↪iteration.
>>> for setting in p.select([0, 1]):
...  pass
>>> for setting in p:
...  print(setting)
f1=a+f2=2
>>> # if volatile is set to True when the selector is set and the setting set␣
↪iterated,
>>> # the setting set is reinitialized at the second iteration.
>>> for setting in p.select([0, 1], volatile=True):
...  pass
>>> for setting in p:
...  print(setting)
f1=a+f2=1
f1=a+f2=2
f1=a+f2=3
f1=b+f2=1
f1=b+f2=2
f1=b+f2=3
f1=c+f2=1
f1=c+f2=2
f1=c+f2=3
>>> # if volatile was set to False (default) when the selector was first set
>>> # and the setting set iterated, the complete set of settings can be reached
>>> # by calling selector with no parameters.
>>> for setting in p.select([0, 1]):
...  pass
>>> for setting in p.select():
...  print(setting)
```

(continues on next page)

(continued from previous page)

```
f1=a+f2=1
f1=a+f2=2
f1=a+f2=3
f1=b+f2=1
f1=b+f2=2
f1=b+f2=3
f1=c+f2=1
f1=c+f2=2
f1=c+f2=3
```

1.6 Setting

Handle the display of settings, the unique description of a parametrization of the system probed by the experiment of the dcode module.

class doce.setting.**Setting**(*plan*, *setting_array=None*, *positional=True*)

stores a *setting*, where each member is a factor and the value of the member is a modality.

Stores a *setting*, where each member is a factor and the value of the member is a modality.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan()
>>> p.f1=['a', 'b']
>>> p.f2=[1, 2]
```

```
>>> for setting in p:
...   print(setting)
f1=a+f2=1
f1=a+f2=2
f1=b+f2=1
f1=b+f2=2
```

Methods

identifier([style, sort, factor_separator, ...])	return a one-liner str or a list of str that describes a setting or a `Plan` object.
perform(function, experiment, log_file_name, ...)	run the function given as parameter for the setting.
remove_factor(factor)	returns a copy of the setting where the specified factor is removed.
replace(factor[, value, positional, relative])	returns a new doce.Plan object with one factor with modified modality.

identifier(*style='long'*, *sort=True*, *factor_separator='+'*, *modality_separator='='*, *singleton=True*, *default=False*, *hide=None*)

return a one-liner str or a list of str that describes a setting or a `Plan` object.

Return a one-liner str or a list of str that describes

a setting or a `Plan` object with a high degree of flexibility.

Parameters

style: str (optional)
'long': (default) 'list': a list of string alternating factor and the corresponding modality 'hash': a hashed version

sort: bool (optional)
if True, sorts the factors by name (default).

If False, use the order of definition.

singleton: bool (optional)
if True, consider factors with only one modality.

if False, consider factors with only one modality (default).

default: bool (optional)
if True, also consider couple of factor/modality where the modality is explicitly set to be a default value for this factor using `doce.Plan.default()`.

if False, do not show them (default).

hide: list of str
list the factors that should not be considered.

factor_separator: str
factor_separator used to concatenate the factors, default is '|'.

factor_separator: str
factor_separator used to concatenate the factor and modality value, default is '='.

See also:

doce.Plan.default
doce.util.compress_name

Examples

```
>>> import doce
```

```
>>> p = doce.Plan()
>>> p.one = ['a', 'b']
>>> p.two = [0,1]
>>> p.three = ['none', 'c']
>>> p.four = 'd'
```

```
>>> print(p)
  0  one: ['a' 'b']
  1  two: [0 1]
  2  three: ['none' 'c']
  3  four: ['d']
```

```
>>> for setting in p.select([0, 1, 1]):
...   # default display
...   print(setting.identifier())
four=d+one=a+three=c+two=1
>>> # list style
>>> print(setting.identifier('list'))
['four=d', 'one=a', 'three=c', 'two=1']
>>> # hashed version of the default display
>>> print(setting.identifier('hash'))
4474b298d3b23000e739e888042dab2b
>>> # do not apply sorting of the factor
>>> print(setting.identifier(sort=False))
one=a+two=1+three=c+four=d
>>> # specify a factor_separator
>>> print(setting.identifier(factor_separator=' '))
four=d one=a three=c two=1
>>> # do not show some factors
>>> print(setting.identifier(hide=['one', 'three']))
four=d+two=1
>>> # do not show factors with only one modality
>>> print(setting.identifier(singleton=False))
one=a+three=c+two=1
>>> delattr(p, 'four')
>>> for setting in p.select([0, 0, 0]):
...   print(setting.identifier())
one=a+three=none+two=0
```

```
>>> # set the default value of factor one to a
>>> p.default('one', 'a')
>>> for setting in p.select([0, 1, 1]):
...   print(setting.identifier())
three=c+two=1
>>> # do not hide the default value in the description
>>> print(setting.identifier(default=True))
one=a+three=c+two=1
```

```
>>> p.optional_parameter = ['value_one', 'value_two']
>>> for setting in p.select([0, 1, 1, 0]):
...   print(setting.identifier())
optional_parameter=value_one+three=c+two=1
>>> delattr(p, 'optional_parameter')
```

```
>>> p.optional_parameter = ['value_one', 'value_two']
>>> for setting in p.select([0, 1, 1, 0]):
...   print(setting.identifier())
optional_parameter=value_one+three=c+two=1
```

perform(*function*, *experiment*, *log_file_name*, **parameters*)

run the function given as parameter for the setting.

Helper function for the method `do()`.

See also:

doce.Plan.do

remove_factor(*factor*)

returns a copy of the setting where the specified factor is removed.

Parameters

factor: str
the name of the factor.

replace(*factor*, *value=None*, *positional=0*, *relative=0*)

returns a new doce.Plan object with one factor with modified modality.

Returns a new doce.Plan object with one factor with modified modality. The value of the requested new modality can requested by 3 exclusive means: its value, its position in the modality array, or its relative position in the array with respect to the position of the current modality.

Parameters

factor: int or str
if int, considered as the index inside an array of the factors sorted by order of definition.

If str, the name of the factor.

modality: literal or None (optional)
the value of the modality.

positional: int (optional)
if 0, this parameter is not considered (default).

If >0, interpreted as the index in the modality array (default).

relative: int (optional)
if 0, this parameter is not considered (default).

Otherwise, interpreted as an index, relative to the current modality.

Examples

```
>>> import doce
```

```
>>> p = doce.Plan()
>>> p.one = ['a', 'b', 'c']
>>> p.two = [1, 2, 3]
```

```
>>> for setting in p.select([1, 1]):
...   # the inital setting
...   print(setting)
one=b+two=2
>>> # the same setting but with the factor 'two' set to modality 1
>>> print(setting.replace('two', value=1))
one=b+two=1
>>> # the same setting but with the first factor set to modality
>>> print(setting.replace(1, value=1))
one=b+two=1
>>> # the same setting but with the factor 'two' set to modality index 0
one=b+two=1
```

(continues on next page)

(continued from previous page)

```
>>> print(setting.replace('two', positional=0))
one=b+two=1
>>> # the same setting but with the factor 'two' set to
>>> # modality of relative index -1 with respect to
>>> # the modality index of the current setting
>>> print(setting.replace('two', relative=-1))
one=b+two=1
```

1.7 Metric

Handle processing of the stored outputs to produce the metrics of the doce module.

class doce.metric.**Metric**

Stores information about the way evaluation metrics are stored and manipulated.

Stores information about the way evaluation metrics are stored and manipulated. Each member of this class describes an evaluation metric and the way it may be abstracted. Two name_spaces (doce.metric.Metric._unit, doce.metric.Metric._description) are available to respectively provide information about the unit of the metric and its semantic.

Each metric may be reduced by any mathematical operation that operate on a vector made available by the numpy library with default parameters.

Two pruning strategies can be complemented to this description in order to remove some items of the metric vector before being abstracted.

One can select one value of the vector by providing its index.

Examples

```
>>> import doce
>>> m = doce.metric.Metric()
>>> m.duration = ['mean', 'std']
>>> m._unit.duration = 'second'
>>> m._description = 'duration of the processing'
```

It is sometimes useful to store complementary data useful for plotting that must not be considered during the reduction.

```
>>> m.metric1 = ['median-0', 'min-0', 'max-0']
```

In this case, the first value will be removed before reduction.

```
>>> m.metric2 = ['median-2', 'min-2', 'max-2', '0%']
```

In this case, the odd values will be removed before reduction and the last reduction will select the first value of the metric vector, expressed in percents by multiplying it by 100.

Methods

get_column_header(plan[, factor_display, ...])	Builds the column header of the reduction setting_description.
name()	Returns a list of str with the names of the metrics.
reduce(settings, path[, setting_encoding, ...])	Apply the reduction directives described in each members of doce.metric.
reduce_from_h5(settings, path[, ...])	Handle reduction of the metrics when considering numpy storage.
reduce_from_npy(settings, path[, ...])	Handle reduction of the metrics when considering numpy storage.

significance_status	

get_column_header(*plan*, *factor_display='long'*, *factor_display_length=2*, *metric_display='long'*, *metric_display_length=2*, *metric_has_data=None*, *reduced_metric_display='capitalize'*)

Builds the column header of the reduction setting_description.

This method builds the column header of the reduction setting_description by formating the Factor names from the doce.Plan class and by describing the reduced metrics.

Parameters

plan
[doce.Plan] The doce.Plan describing the factors of the experiment.

factor_display
[str (optional)] The expected format of the display of factors. 'long' (default) do not lead to any reduction. If factor_display contains 'short', a reduction of each word is performed.

- 'short_underscore' assumes python_case delimitation.
- 'short_capital' assumes camel_case delimitation.
- 'short' attempts to perform reduction by guessing the type of delimitation.

factor_display_length
[int (optional)] If factor_display has 'short', factor_display_length specifies the maximal length of each word of the description of the factor.

metric_has_data
[list of bool] Specify for each metric described in the doce.metric.Metric object, whether data has been loaded or not.

reduced_metric_display
[str (optional)] If set to 'capitalize' (default), the description of the reduced metric is done in a Camel case fashion: metricReduction.

If set to 'underscore', the description of the reduced metric is done in a snake case fashion: metric_reduction.

See also:

doce.util.compress_description

name()

Returns a list of str with the names of the metrics.

Returns a list of str with the names of the metrics defined as members of the doce.metric.Metric object.

Examples

```
>>> import doce
>>> m = doce.metric.Metric()
>>> m.duration = ['mean']
>>> m.mse = ['mean']
>>> m.name()
['duration', 'mse']
```

reduce(*settings*, *path*, *setting_encoding=None*, *factor_display='long'*, *factor_display_length=2*, *metric_display='long'*, *metric_display_length=2*, *reduced_metric_display='capitalize'*, *verbose=False*)

Apply the reduction directives described in each members of doce.metric. Metric objects for the settings given as parameters.

For each setting in the iterable settings, available data corresponding to the metrics specified as members of the doce.metric.Metric object are reduced using specified reduction methods.

Parameters

settings: doce.Plan
iterable settings.

path: str
In the case of .npy storage, a valid path to the main directory. In the case of .h5 storage, a valid path to an .h5 file.

setting_encoding
[dict] Encoding of the setting. See doce.Plan.id for references.

reduced_metric_display
[str (optional)] If set to 'capitalize' (default), the description of the reduced metric is done in a Camel case fashion: metric_reduction.

If set to 'underscore', the description of the reduced metric is done in a Python case fashion: metric_reduction.

factor
[doce.Plan] The doce.Plan describing the factors of the experiment.

factor_display
[str (optional)] The expected format of the display of factors. 'long' (default) do not lead to any reduction. If factor_display contains 'short', a reduction of each word is performed.

- 'short_underscore' assumes python_case delimitation.
- 'short_capital' assumes camel_case delimitation.
- 'short' attempts to perform reduction by guessing the type of delimitation.

factor_display_length
[int (optional)] If factor_display has 'short', factor_display_length specifies the maximal length of each word of the description of the factor.

verbose
[bool] In the case of .npy metric storage, if verbose is set to True, print the file_name seeked for each metric as well as its time of last modification.

In the case of .h5 metric storage, if verbose is set to True, print the group seeked for each metric.

Returns

setting_description
[list of lists of literals] A setting_description, stored as a list of list of literals of the same size. The main list stores the rows of the setting_description.

column_header
[list of str] The column header of the setting_description as a list of str, describing the factors (left side), and the reduced metrics (right side).

constant_setting_description
[str] When a factor is equally valued for all the settings, the factor column is removed from the setting_description and stored in constant_setting_description along its value.

nb_column_factor
[int] The number of factors in the column header.

Examples

doce supports metrics storage using an .npy file per metric per setting.

```
>>> import doce
>>> import numpy as np
>>> import pandas as pd
>>> np.random.seed(0)
```

```
>>> experiment = doce.experiment.Experiment()
>>> experiment.name = 'example'
>>> experiment.set_path('output', '/tmp/'+experiment.name+'/', force=True)
>>> experiment.add_plan('plan', f1 = [1, 2], f2 = [1, 2, 3])
>>> experiment.set_metric(name = 'm1_mean', output = 'm1', func = np.mean)
>>> experiment.set_metric(name = 'm1_std', output = 'm1', func = np.std)
>>> experiment.set_metric(name = 'm2_min', output = 'm2', func = np.min)
>>> experiment.set_metric(name = 'm2_argmin', output = 'm2', func = np.argmin)
>>> def process(setting, experiment):
...   metric1 = setting.f1+setting.f2+np.random.randn(100)
...   metric2 = setting.f1*setting.f2*np.random.randn(100)
...   np.save(experiment.path.output+setting.identifier()+'_m1.npy', metric1)
...   np.save(experiment.path.output+setting.identifier()+'_m2.npy', metric2)
>>> nb_failed = experiment.perform([], process, progress='')
>>> (setting_description,
... column_header,
... constant_setting_description,
... nb_column_factor,
... modification_time_stamp,
... p_values
... ) = experiment.metric.reduce(experiment._plan.select([1]), experiment.path)
```

```
>>> df = pd.DataFrame(setting_description, columns=column_header)
>>> df[column_header[nb_column_factor:]] = df[column_header[nb_column_factor:]].
↪round(decimals=2)
>>> print(constant_setting_description)
f1: 2
>>> print(df)
  f2  m1_mean  m1_std  m2_min  m2_argmin
0   1    2.87    1.00  -4.49         35
1   2    3.97    0.93  -8.19         13
2   3    5.00    0.91 -12.07         98
```

doce also supports metrics storage using one .h5 file sink structured with settings as groups et metrics as leaf nodes.

```
>>> import doce
>>> import numpy as np
>>> import tables as tb
>>> import pandas as pd
>>> np.random.seed(0)
```

```
>>> experiment = doce.experiment.Experiment()
>>> experiment.name = 'example'
>>> experiment.set_path('output', '/tmp/'+experiment.name+'.h5', force=True)
>>> experiment.add_plan('plan', f1 = [1, 2], f2 = [1, 2, 3])
>>> experiment.set_metric(name = 'm1_mean', output = 'm1', func = np.mean)
>>> experiment.set_metric(name = 'm1_std', output = 'm1', func = np.std)
>>> experiment.set_metric(name = 'm2_min', output = 'm2', func = np.min)
>>> experiment.set_metric(name = 'm2_argmin', output = 'm2', func = np.argmin)
>>> def process(setting, experiment):
...   h5 = tb.open_file(experiment.path.output, mode='a')
...   setting_group = experiment.add_setting_group(
...     h5,
...     setting,
...     output_dimension = {'m1':100, 'm2':100}
...   )
...   setting_group.m1[:] = setting.f1+setting.f2+np.random.randn(100)
...   setting_group.m2[:] = setting.f1*setting.f2*np.random.randn(100)
...   h5.close()
>>> nb_failed = experiment.perform([], process, progress='')
>>> h5 = tb.open_file(experiment.path.output, mode='r')
>>> print(h5)
/tmp/example.h5 (File) ''
Last modif.: '...'
    Object Tree:
/ (RootGroup) ''
/f1=1+f2=1 (Group) 'f1=1+f2=1'
/f1=1+f2=1/m1 (Array(100,)) 'm1'
/f1=1+f2=1/m2 (EArray(100,)) 'm2'
/f1=1+f2=2 (Group) 'f1=1+f2=2'
/f1=1+f2=2/m1 (Array(100,)) 'm1'
/f1=1+f2=2/m2 (EArray(100,)) 'm2'
/f1=1+f2=3 (Group) 'f1=1+f2=3'
/f1=1+f2=3/m1 (Array(100,)) 'm1'
```

(continues on next page)

(continued from previous page)

```
/f1=1+f2=3/m2 (EArray(100,)) 'm2'
/f1=2+f2=1 (Group) 'f1=2+f2=1'
/f1=2+f2=1/m1 (Array(100,)) 'm1'
/f1=2+f2=1/m2 (EArray(100,)) 'm2'
/f1=2+f2=2 (Group) 'f1=2+f2=2'
/f1=2+f2=2/m1 (Array(100,)) 'm1'
/f1=2+f2=2/m2 (EArray(100,)) 'm2'
/f1=2+f2=3 (Group) 'f1=2+f2=3'
/f1=2+f2=3/m1 (Array(100,)) 'm1'
/f1=2+f2=3/m2 (EArray(100,)) 'm2'
>>> h5.close()
```

```
>>> (setting_description,
... column_header,
... constant_setting_description,
... nb_column_factor,
... modification_time_stamp,
... p_values) = experiment.metric.reduce(experiment.plan.select([0]),␣
↪experiment.path)
```

```
>>> df = pd.DataFrame(setting_description, columns=column_header)
>>> df[column_header[nb_column_factor:]] = df[column_header[nb_column_factor:]].
↪round(decimals=2)
>>> print(constant_setting_description)
f1: 1
>>> print(df)
  f2  m1_mean  m1_std  m2_min  m2_argmin
0  1    2.06    1.01  -2.22         83
1  2    2.94    0.95  -5.32         34
2  3    3.99    1.04  -9.14         89
```

reduce_from_h5(*settings*, *path*, *setting_encoding=None*, *verbose=False*)

Handle reduction of the metrics when considering numpy storage.

The method handles the reduction of the metrics when considering h5 storage.

The method *doce.metric.Metric.reduce()* wraps this method and should be considered as the main user interface, please see its documentation for usage.

See also:

doce.metric.Metric.reduce

reduce_from_npy(*settings*, *path*, *setting_encoding=None*, *verbose=False*)

Handle reduction of the metrics when considering numpy storage.

The method handles the reduction of the metrics when considering numpy storage. For each metric, a .npy file is assumed to be available which the following naming convention: <id_of_setting>_<metric_name>.npy.

The method *doce.metric.Metric.reduce()* wraps this method and should be considered as the main user interface, please see its documentation for usage.

See also:

doce.metric.Metric.reduce

1.8 Util

Handle low level functionalities of the doce module.

doce.util.**compress_description**(*description*, *desc_type='long'*, *atom_length=2*)

reduces the number of letters for each word in a given description structured with underscores (python_case) or capital letters (camel_case).

Parameters

description
[str] the structured description.

desc_type
[str, optional] can be 'long' (default), do not lead to any reduction, 'short_underscore' assumes python_case delimitation, 'short_capital' assumes camel_case delimitation, and 'short' attempts to perform reduction by guessing the type of delimitation.

Returns

compressed_description
[str] The compressed description.

Examples

```
>>> import doce
>>> doce.util.compress_description(
... 'myVeryLongParameter',
... desc_type='short'
... )
'myvelopa'
>>> doce.util.compress_description(
...  'that_very_long_parameter',
...  desc_type='short',
...  atom_length=3
...  )
'thaverlonpar'
```

doce.util.**constant_column**(*table=None*)

detect which column(s) have the same value for all lines.

Parameters

table
[list of equal size lists or None] table of literals.

Returns

values
[list of literals] values of the constant valued columns, None if the column is not constant.

Examples

```
>>> import doce
>>> table = [['a', 'b', 1, 2], ['a', 'c', 2, 2], ['a', 'b', 2, 2]]
>>> doce.util.constant_column(table)
['a', None, None, 2]
```

doce.util.**in_notebook**()

detect if the experiment is running from Ipython notebook.

doce.util.**prune_setting_description**(*setting_description*, *column_header=None*, *nb_column_factor=0*, *factor_display='long'*, *show_unique_setting=False*)

remove the columns corresponding to factors with only one modality from the setting_description and the column_header.

Remove the columns corresponding to factors with only one modality from the setting_description and the column_header and describes the factors with only one modality in a separate string.

Parameters

setting_description: list of list of literals
the body of the table.

column_header: list of string (optional)
the column header of the table.

nb_column_factor: int (optional)
the number of columns corresponding to factors (default 0).

factor_display:
type of description of the factors (default 'long'), see *doce.util.compress_description()* for reference.

show_unique_setting: bool
If True, show the description of the unique setting in cst_setting_desc.

Returns

setting_description: list of list of literals
setting_description where the columns corresponding to factors with only one modality are removed.

column_header: list of str
column_header where the columns corresponding to factors with only one modality are removed.

cst_setting_desc: str
description of the settings with constant modality.

nb_column_factor: int
number of factors in the new setting_description.

Examples

```
>>> import doce
```

```
>>> header = ['factor_1', 'factor_2', 'metric_1', 'metric_2']
>>> table = [['a', 'b', 1, 2], ['a', 'c', 2, 2], ['a', 'b', 2, 2]]
>>> (setting_description,
...  column_header,
...  cst_setting_desc,
...  nb_column_factor) = doce.util.prune_setting_description(table, header, 2)
>>> print(nb_column_factor)
1
>>> print(cst_setting_desc)
factor_1: a
>>> print(column_header)
['factor_2', 'metric_1', 'metric_2']
>>> print(setting_description)
[['b', 1, 2], ['c', 2, 2], ['b', 2, 2]]
```

doce.util.**query_yes_no**(*question*, *default='yes'*)

ask a yes/no question via input() and return their answer.

The 'answer' return value is True for 'yes' or False for 'no'.

Parameters

question
[str] phrase presented to the user.

default
[str or None (optional)] presumed answer if the user just hits <Enter>. It must be 'yes' (default), 'no' or None. The latter meaning an answer is required of the user.

Returns

answer
[bool] True if prompt is 'yes'.

False if prompt is 'no'.

1.9 Changelog

1.9.1 v0.1

v0.1.0

2020-07-22

New Features

- test

1.10 Index

:ref:genindex

1.11 Glossary

system:
A system is a computational process that can be controled by a fixed set of parameters, whose execution can be reliably replicated.

experiment:
An experiment is the environnement within which the system is operated. It comprises the data, the experimental code, and the set of factors required to operate the system.

factor
A factor is a degree of freedom in the design of the system. In doce, it is implemented as a list of modalities.

modality
A modality is an instantiation of a factor in a setting.

setting
A setting is a list of modalities, that fully describes the parameters of the system, where each modality is issued from each factor of the experiment.

selector
A selector is a convenient way to define a set of settings. In doce, it is alternatively expressed as a list of dict or a list of list. In the former, each list is composed of dict, each with the following syntax: {factor: modality or list of modalities, ...}. In the latter, each list is composed of integer values or list of integer values. For example, the selector [0, -1, [1, 2]] defines the set of setting with the first modality of the first factor, all the modalities of the second factor, and the second and third modality of the third factor.

metric
A metric is a set of data that results from the execution of the system given a setting. Each metric can de reduced in order to produce quantities that will be useful for monitoring the behaviour of the system.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

C

D

E

F

G

I

M

N

P

Q

R

S