

PyExperimentSuite Documentation

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 2 | Download and Installation | 2 |
| 2.1 | Requirements | 2 |
| 2.2 | Download from github.com | 3 |
| 2.3 | Download with git | 4 |
| 2.4 | Successful Installation | 4 |
| 3 | Using PyExperimentSuite | 4 |
| 3.1 | Typical Workflow | 5 |
| 3.2 | Getting Started | 5 |
| 3.3 | Implementing reset() and iterate() | 7 |
| 3.4 | Optional Restore Functionality | 10 |
| 3.5 | PyExperimentSuite for non-Python experiments | 12 |
| 4 | The Configuration File | 13 |
| 4.1 | Basic Parameter Definitions | 14 |
| 4.2 | Evaluating Parameter Ranges | 15 |
| 4.3 | Parameter Range Combinations | 16 |
| 5 | Run-Time Options | 17 |
| 5.1 | Help | 17 |
| 5.2 | Configuration File | 18 |
| 5.3 | Number of Cores | 18 |
| 5.4 | Deletion of Old Experiments | 18 |
| 5.5 | Selected Experiments | 18 |
| 5.6 | Browse Experiments | 19 |
| 5.7 | Progress of Experiments | 19 |
| 6 | Result Retrieval | 20 |
| 6.1 | Log Files | 20 |

| | | |
|----------|--|-----------|
| 6.2 | Python Interface | 21 |
| 6.3 | Retrieving Information about Experiments | 22 |
| 6.4 | Single Values | 22 |
| 6.5 | Single Values from Parameter Range Experiments | 23 |
| 6.6 | Histories | 24 |
| 6.7 | Histories from Parameter Range Experiments | 25 |
| 6.8 | Aggregated Histories over Repetitions | 25 |
| 7 | Examples and Best Practices | 26 |
| 7.1 | Strings in Iterable Objects | 26 |
| 7.2 | Passing Classes as Parameters | 27 |
| 7.3 | Passing Lists as Parameters | 28 |
| 7.4 | Debugging PyExperimentSuite Scripts | 28 |

1 Introduction

PyExperimentSuite is an open source software tool written in Python, that supports scientists, engineers and others to conduct automated software experiments on a large scale with numerous different parameters. It reads parameters (or ranges of parameters) from a configuration file, runs the experiments using multiple cores if desired and logs the results in files. Parameter combinations can be evaluated as a grid (each combination of parameters) or in a list (try several defined parameter combinations in row). PyExperimentSuite also supports continuing any experiments where left off when the execution was interrupted (e.g. power failure, process was killed, etc.). The experiment results can be obtained in different ways by a built-in Python interface.

2 Download and Installation

To install PyExperimentSuite, you can either download the package from the Github website, or install it directly with *git*. If you're not familiar with the version management command line tool *git*, follow the steps in section 2.2.

2.1 Requirements

PyExperimentSuite is developed in Python and does not have many dependencies. It does require Python 2.6 or higher, though, because it makes use of the multiprocessing-

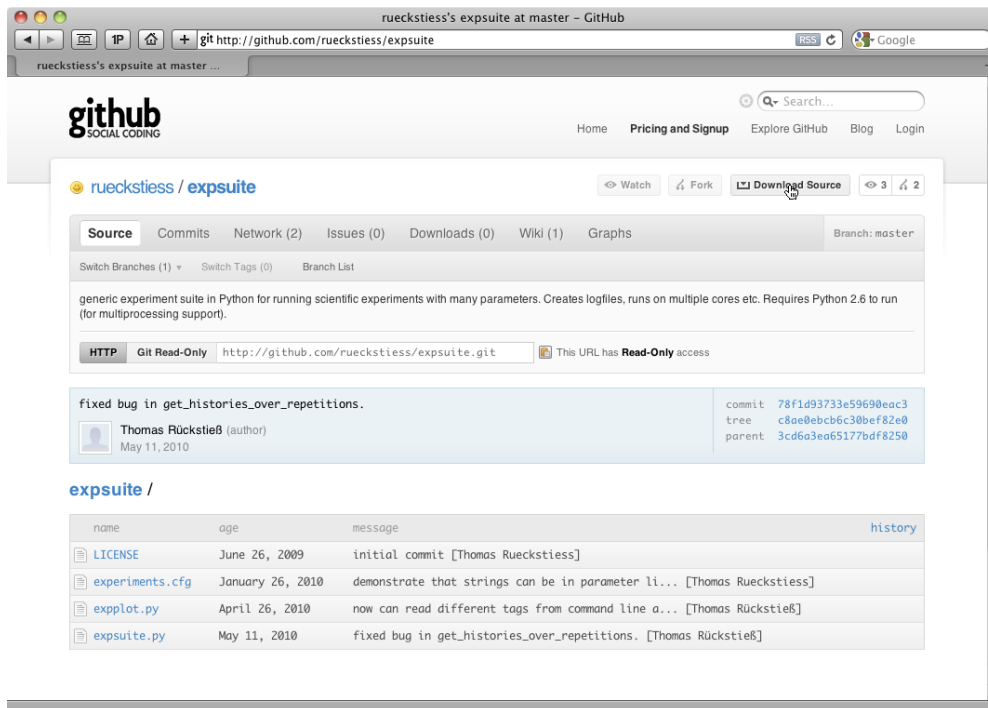


Figure 1: Downloading PyExperimentSuite from github.com. Click on the button “Download Source” on the top right and then select the type of archive you would like to download.

ing module, which was only added in version 2.6. Another dependency is numpy. Other than that, PyExperimentSuite is a stand-alone package.

2.2 Download from github.com

To download and setup PyExperimentSuite manually, go to the the website on which the project is hosted: <http://github.com/rueckstiess/expsuite/>. Click on the “Download Source” button on the top right and select the type of archive (zip or tar) that you would like to download (see Fig. 1).

After the download is complete, unpack the archive and move the extracted folder to your desired location, preferably in a directory of your `$PYTHONPATH` environment variable (otherwise, you might have to update your `$PYTHONPATH` environment variable to include the destination path).

2.3 Download with git

If you are familiar with `git` and have it installed on your system, you can simply clone the project directly from github. Switch to the directory under which you want to install the `expsuite` directory. This directory should be in your `$PYTHONPATH` environment variable (otherwise, you might have to update your `$PYTHONPATH` environment variable to include the destination path). Now run this command from the command line:

```
$ git clone http://github.com/rueckstiess/expsuite.git
```

A directory will be created under the current directory and the sources will be downloaded into the directory. Note that you will not be able to push any committed changes back to the github repository, it is read-only.

2.4 Successful Installation

To test, whether the installation was successful, open your Python interpreter by typing `python` at the command line, and import the package like this:

```
>>> import expsuite
```

If you receive no error messages, installation was successful and you can now use `PyExperimentSuite`.

3 Using PyExperimentSuite

`PyExperimentSuite` is most useful for experiments written in Python. The next sections will demonstrate, how you derive from the `PyExperimentSuite` base class to create your own suite, how you implement the necessary functions `reset()` and `iterate()`, and what the restore option on iteration level is and how to make use of it.

If your experiments are not written in Python but you still want to try out `PyExperimentSuite`, you find some help in Section 3.5.

3.1 Typical Workflow

We will give the common workflow with `PyExperimentSuite` here at the beginning as an overview of what lies ahead of you. These steps are then explained in detail in the following sections.

In order to successfully use `PyExperimentSuite`, these steps need to be performed:

1. create a class which derives from the `PyExperimentSuite` class
2. add the `if __name__ == '__main__':` part at the bottom of the file
3. fill out the `reset()` and `iterate()` methods
4. (optional) fill out the `save_state()` and `restore_state()` methods and set the `restore_supported` flag to `True`
5. create or edit the `experiments.cfg` file and add all the parameters and experiments
6. run the suite from the command line
7. after completion, open your Python console or create a Python script and use the built-in interface to access your results and visualize or post-process them

3.2 Getting Started

Each new experiment setup requires that you write a few lines of Python code. You need to create a new class that inherits from the `PyExperimentSuite` base class, fill in the missing `reset()` and `iterate()` methods, and optionally take care of saving and loading relevant data for your experiments so the suite can be interrupted and continued at any time.

It is recommended (but not essential), that you create a new folder for your experiment setup. `PyExperimentSuite` will create subfolders below its execution folder and depending on how many experiments you create, it is easy to loose oversight of all your experiments. Therefore, it is best to keep everything belonging to one set of experiments in one folder.

For a typical experiment setup, create a new file and give it a plausible name, for example `suite.py`. Import the `PyExperimentSuite` class and create a new class that inherits from `PyExperimentSuite`. Next, add the empty method declarations for

the two functions `reset()` and `iterate()` as shown in Listing 1. While `reset()` does not need a return value, `iterate()` needs to return a dictionary. For now, this can be the empty dictionary. Finally, add the three lines at the bottom of the script for creating the `MySuite` object and starting it. If you're done with this task, you have completed workflow point 1 and 2 from Section 3.1.

Listing 1: `MySuite` class definition with empty methods for later implementation.

```
from expsuite import PyExperimentSuite

class MySuite(PyExperimentSuite):

    def reset(self, params, rep):
        pass

    def iterate(self, params, rep, n):
        ret = {}
        return ret

if __name__ == '__main__':
    mysuite = MySuite()
    mysuite.start()
```

If you go to your command line and start the script with

```
$ python suite.py
```

an error message will appear to notify you that the config file `experiments.cfg` could not be found. To remedy this, create a second file called `experiments.cfg` in the same folder and add a default section with some parameters, as shown in Listing 2. These three parameters are necessary in all your configuration files and `PyExperimentSuite` will complain, if any of them are missing.

Listing 2: Configuration file with default section.

```
[DEFAULT]

repetitions = 1
iterations = 100
path = results
```

After saving the configuration file, go back to the command line and try running the script again:

```
$ python suite.py
```

This time, there should be no error message after hitting enter. The script executes all the defined experiments (which is currently *none*) and returns to the command line.

If you want to see a little more feedback from your script, run the help option on the command line with

```
$ python suite.py --help
```

Now you receive a short list of possible commands that your script can be executed with.

3.3 Implementing `reset()` and `iterate()`

The main implementation work you have to do for `PyExperimentSuite` goes in the two functions `reset()` and `iterate()`. In order to understand better what they do, have a look at Listing 3. It demonstrates in a simplified code example the execution order of these functions.

Listing 3: Simplified execution loop of an experiment in the suite.

```
# execute this loop for each experiment
for r in range(repetitions):
    reset()
    for i in range(iterations):
        iterate()
```

For each new experiment, the `reset()` method is called first. Here, all the necessary objects should be created that are required by the experiment. Everything needed for the experiment should be assigned to class variables (with the prefix `self.`) because it is needed in the `iterate()` method later on, which doesn't have access to local variables.

Implementation of `reset()`

What code exactly goes into your `reset()` method depends entirely on the experiment you want to set up.

Let's say you want to train a Neural Network¹ with a given dataset and test several different learning rates as part of your experiment. Your `reset()` method will need the code to initialize the Neural Network (perhaps with random weights), tell the network where it can find the dataset and also set the learning rate. Listing 4 shows how this would be implemented in the `reset()` function. Note that you have to import all the necessary modules and classes before, the listing only shows an example for the `reset()` definition.

Listing 4: Example of `reset()` method for training a Neural Network

```
def reset(self, params, rep):

    # create network and randomize weights
    self.network = NeuralNetwork()
    self.network.randomizeWeights()

    # give dataset to network
    path = os.path.join('./data/', params['filename'])
    self.network.setDataSet(path)

    # set learning rate
    self.network.learningrate = params['learningrate']
```

The network is assigned to a class variable `self.network` because we will need access to it later on in the `iterate()` method. Values that are not required any further can be stored in local variables, like the `path` of the dataset in our example. In this example we make both the filename and the learning rate a parameter of the experiment, which needs to be defined in the configuration file `experiments.cfg`.

We can then access these parameters through the dictionary `params` which uses the parameter name defined in `experiments.cfg` as key. This means, that we need two more parameters in our configuration file. Listing 5 shows the changes to the default section of our original config file from Section 3.2.

In addition to the `params` dictionary, that is passed to `reset()` and contains the parameters of the current experiment, there is a second parameter `rep`. This integer contains the index of the current repetition, starting from 0 and increasing for each repetition, up to `params['repetitions']-1`. Sometimes this number might be of use to you, for example if you have to load a particular datafile for each separate repetition, or if the initialization of each experiment needs to be unique. You could for

¹You don't need to know what a Neural Network is or how it works in order to understand this example. Just think of it as a black box that has some internal parameters (called *weights*) that need to be adjusted according to a given dataset, which is presented to the network repeatedly. This process is called *training*. It further contains a free parameter called *learning rate* that determines how fast the network will learn. Finding the optimal learning rate is not trivial and is often done by trial-and-error.

example seed a random number generator with a combination of the system time and this repetition number, to guarantee a unique random sequence for each repetition.

Listing 5: Configuration file for the neural network example.

```
[DEFAULT]

repetitions = 1
iterations = 100
path = results

filename = 'stockmarket.data'
learningrate = 0.01
```

Implementation of `iterate()`

The `iterate()` method will be called repeatedly during the execution of one single repetition of one experiment. It is a single step or cycle within the experiment. Most experiments have a natural separation into repeated iterations already. If your experiment does not seem to be iterative, you might have to introduce some separation into steps manually. If you feel like this is not possible at all, you can also use a single iteration for the whole experiment, setting the `iterations` parameter in the config file to 1.

Just as in the `reset()` method, the `params` dictionary is also passed to the `iterate()` method. The second parameter passed to this method is `rep`, which indicates the current index of repetition, starting from 0. `iterate()` has a third parameter, `n`, which represents the current iteration index. The variable `n` starts from 0 and counts up to `params['iterations']-1`.

While the `reset()` method did not return any values, the `iterate()` method is expected to return a dictionary. The dictionary should contain the relevant information, that you want to store in the log files. This can be some measure of progress of your experiment, an error, some identifying strings or the running index `n`.

Put the relevant values, together with a descriptive key, into a dictionary and return it at the end of the function:

```
ret = {'first_value':value1, 'another_value':value2}
return ret
```

Make sure that none of the keys contains spaces, because the key-value pairs them-

selves are stored with spaces as delimiters in the log files. If you choose a key that contains spaces by accident, PyExperimentSuite will rename it and replace the spaces with underscores, and issue a warning.

More about how to retrieve this information after your experiments have finished can be found in Section 6.

To continue the Neural Network example, we will now execute a learning step in the `iterate()` function. We assume that our `NeuralNetwork` class offers two methods, `train()` and `test()`. The first will execute a training step in the network and return the training error. The latter then runs the network on an independent test dataset and evaluates the test error. We'd like to store both values, together with the current iteration step, in the log files. Listing 6 demonstrates how to achieve this. The keys in the dictionary are strings, with which we will later be able to retrieve the results. Use unique, descriptive labels for these keys.

Listing 6: Example of `iterate()` method for training a Neural Network

```
def iterate(self, params, rep, n):  
  
    trainerr = self.network.train()  
    testerr = self.network.test()  
  
    ret = {'n':n, 'trainerror':trainerr, 'testerror':testerr}  
  
    return ret
```

The suite is now ready to execute the experiments. By calling the script from the command line, it parses the configuration file, runs the required number of experiments, repetitions and iterations, and stores the results in log files.

At this point you have completed step 3 of the workflow checklist from Section 3.1.

3.4 Optional Restore Functionality

So far, we implemented everything necessary to successfully run the PyExperimentSuite. Repetitions will be spread over the available cores on your machine, speeding up the execution by a factor of how many cores you have (and be willing to share with PyExperimentSuite).

If you interrupt the suite while it is executing the experiments, by killing the process or even by a power failure, the suite is left in a somewhat undefined state.

Without any further implementation work from you, all it can do is to abandon the already executed iterations in the current repetition and start the repetition again. PyExperimentSuite does that automatically: it will delete the already logged iterations of the current repetition (or repetitions, if you use multiple cores) and restart them from iteration step 0, by calling the `reset()` method again. Already completed repetitions are of course not affected and will remain in the log files.

Still, some experiments can be very time-consuming and even losing a few iteration steps might be unacceptable to you. In this case, you need to set the class variable `restore_supported` to `True` and implement a few more lines of code, namely the two methods `save_state()` and `restore_state()`. The `save_state()` method is called after each iteration step has completed, and its task is to save all relevant information needed to continue from this iteration step to the disk. This can be done by pickling² crucial objects to a file, or otherwise saving the current state of the experiment. The current repetition and iteration indices are again passed to the function as variables `rep` and `n`. This might be useful if you want to store not only the current iteration state but perhaps want to create a history of all iteration steps. The suite can easily be extended to start from any particular point in time, not just the last one.

If the variable `restore_supported` is set to `True`, the suite checks upon start if there are unfinished experiments on iteration level. It then calls the `restore_state()` method with the appropriate parameters, repetition and iteration index. The task of the `restore_state` function is to load the correct data from files and restore the class objects to that state. Before the `restore_state` function is called, a call to `reset()` has already been made. All you have to do is change the initialized class members to the values you saved in the `save_state()` method.

Let's continue the NeuralNetwork example and fill out the two methods, shown in Listing 7. First of all, we need to enable the `restore_supported` flag, so the PyExperimentSuite knows that it should call the appropriate functions to save and restore the data. Next, we implement the `save_state()` method. We create a file handle for file write access. The location of the file can be anywhere, but it is probably best to place it in the current experiment's folder, where the log files are stored. This location can be retrieved by concatenating the parameters' `path` and `name` fields. `os.path.join()` is perfectly suited for this. As we might execute several repetitions simultaneously on different cores, it is absolutely necessary to have one file per repetition, thus using the repetition index `rep` in the file name. In this example we only save the last executed iteration per repetition, so we don't need to create separate files for each iteration. We use the module `cPickle` to serialize the whole object, dump it in the file, and close the file afterwards.

²`pickle`, and its faster C implementation `cPickle` are Python packages that can serialize objects and save them to disk, and restore the files and create objects again. See the Python documentation for more information about these modules.

Restoring the data is straight-forward as well. Instead of write access, we create a file handle with read access to the file we want to restore. Again, the repetition index is necessary as we have one file for each repetition. Using `cPickle`'s `load()` function, we import the stored data and assign it to the class variable `self.network`.

That's all the necessary steps and you have completed workflow item 4 from the list in Section 3.1. If you run the script now, and abort or kill it during runtime, the most data you could lose is one single iteration (the one that was currently being calculated, before the results could have been saved to disk).

When starting `PyExperimentSuite` again, it will load the last available iteration and continue from where it left off.

Listing 7: Save and restore methods for the Neural Network example.

```
restore_supported = True

def save_state(self, params, rep, n):
    f = open(os.path.join(params['path'], params['name'],
                          'network_%i.saved'%rep), 'w')
    cPickle.dump(f, self.network)
    f.close()

def restore_state(self, params, rep, n):
    f = open('os.path.join(params['path'], params['name'],
                          'network_%i.saved'%rep), 'r')
    self.network = cPickle.load(f)
    f.close()
```

3.5 PyExperimentSuite for non-Python experiments

Although `PyExperimentSuite` is most useful for your Python experiments, it is possible to use it for all kinds of automated scripts, even in other languages. The only condition would be, that the experiment can be run from the command line. This includes any C programs, shell scripts or other tools that you might want to evaluate.

In order to make them work with `PyExperimentSuite`, you need to make use of Python's command line modules, like `sys`, `os` and others. Basically, you need to call the necessary scripts from within Python, and find a way to grab the return value. A good possibility might be the function `os.open()` or `os.popen3()`, which lets you access `stdin`, `stdout` and `stderr` streams as file handles.

This feature has not been tested much and might not work in all environments. I would therefore classify this feature as experimental for now. Feedback on successful usage of non-Python experiments is more than welcome.

4 The Configuration File

So far, we have talked a lot about how to set up your python suite script and how to implement the necessary methods. This Section will give more details about the configuration file which is workflow step 5 from Section 3.1.

The default configuration file is called `experiments.cfg` and is expected to sit in the same folder as your `suite.py` script (or whatever name you choose for it instead). The configuration script is parsed by the python module `ConfigParser`³ and follows its syntax.

Basically, the file is separated in different sections, that start with a section header in square brackets, like `[sectionname]`. Following the section header are entries of the form `name:value` or `name=value`, each on a single line. Leading whitespace is removed from values. Lines beginning with `#` or `;` are ignored and can be used as comments.

A special section with header `[DEFAULT]` can be provided. Any other section will inherit all key-value pairs from this section, unless it defines another value for an existing key, in which case the new value will be chosen instead.

For the `PyExperimentSuite`, it is recommended to create one `[DEFAULT]` section for all parameters that are common to all experiments (like the path to store the results for example), and to create a separate section for each different experiment. The three keys `repetitions`, `iterations` and `path` need to exist in all experiments, so it makes sense to define default values for them in the `[DEFAULT]` section.

There is one more parameter that takes a special role for `PyExperimentSuite`, and that is `experiment`. We will discuss its relevance further down in this section. All other parameters are entirely dependent on the kind of experiments you want to conduct and need to be chosen accordingly.

³<http://docs.python.org/library/configparser.html>

4.1 Basic Parameter Definitions

Acceptable values include any type that can be evaluated to a Python int, float, string, or object. In fact, PyExperimentSuite will try to evaluate the given value using the Python function `eval()`, and if no errors occur, interpret it as its Python pendant. This means, that everything that can be evaluated to an integer will be treated as one, the same is true for floats. More complex values, like `sin(2.)` or `0.5*pi` are also valid expressions.

Everything, that cannot be evaluated without error, will be interpreted as a string. This means that both strings with and without quotation marks will be interpreted as a string in Python. It is your choice to add quotation marks (single or double) or not. If you do, they will be stripped away.

Lists and iterable objects play a special role as explained in 4.2 below and will, by default, not be passed to the parameters as their Python equivalent.

Listing 8 demonstrates some valid key-value pairs.

Listing 8: Examples for valid key-value pairs for the configuration file.

```
[DEFAULT]
path = ./results/
repetitions = 1
iterations = 100

filename = 'stockmarket.data'
learningrate = 0.01

[experiment1]
path = "./testruns/"
alpha = 0.1
beta = 2.
gamma = sin(2.)
delta = 0.5*pi

[experiment2]
comment = "This is experiment 2"
alpha = 0.1
beta = [0.5, 1.0, 1.5]

[experiment3]
comment = "Experiment 3 uses a grid for the parameter combinations"
experiment = 'grid'
```

```

alpha = arange(0.0, 1.0, 0.2)
beta = [0.5, 1.0, 2.0]

[experiment4]
comment = "Experiment 4 uses a list for the parameter combinations."
experiment = list
alpha = arange(0.0, 1.0, 0.2)
beta = [0.5, 1.0, 2.0]

[experiment5]
comment = "Experiment 5 will assign the whole lists to parameters."
experiment = 'single'
alpha = arange(0.0, 1.0, 0.2)
beta = [0.5, 1.0, 2.0]

```

4.2 Evaluating Parameter Ranges

If your value evaluates to a python object that is iterable (e.g. lists, numpy arrays, generator objects, etc.) PyExperimentSuite will not assign the object to the parameter by default. Instead, it assumes that you want to try a range of values in separate experiments. This is very handy if you need to find the optimal value for certain parameters, because you can just define the range and let PyExperimentSuite do all the testing. If you need to assign a list or iterable object to a parameter and do not want PyExperimentSuite to create several parameter range experiments in this way, you need to set your experiment type to `'single'`. Section 7.3 explains how to do that. Per default, the elements of iterable objects are passed to the parameter individually in separate sub-experiments.

Log files are usually stored in a directory named `./path/name/` where `path` is the path parameter from the configuration file and `name` is the section name of the experiment. If you define a parameter as a range, PyExperimentSuite will create another level of sub-directories under `./path/name/`. Each sub-directory has a unique name that consists of the parameter names and values of the ranges.

Experiment 2 from Listing 8 would create the following three paths:

```

./results/experiment2/beta0.5/
./results/experiment2/beta1.0/
./results/experiment2/beta1.5/

```

This also helps identify the folders of certain experiments, if you should ever look for the log files. In the normal use case, you will most likely not have to go and find these

files by yourself but use the Python interface to retrieve your results instead. More about the interface can be found in Section 6.

All other non-iterable parameters are kept constant for the experiments. This is also true for the `repetitions` parameter, which means that each of these “sub-experiments” are repeated an equal number of times, defined by `repetitions` in the config file.

4.3 Parameter Range Combinations

Should you define more than one parameter in your config file that evaluates to an iterable object, you have two choices: try every single combination of all parameter choices or run successive experiments with the parameters of the same list indices. Figure 2 explains the difference graphically.

The first choice is called a *grid* experiment, and is also set as the default. `PyExperimentSuite` will create sub-experiments for every single combination of all parameters.

Experiment 3 in Listing 8 defines two iterable parameters, `alpha` and `beta`. Numpy’s `arange()` evaluates to a list with the values 0.0, 0.2, 0.4, 0.6 and 0.8 for `alpha`, and `beta` takes on the values of 0.5, 1.0 and 2.0. In total, `PyExperimentSuite` will run $5 \cdot 3 = 15$ experiments, and the folders for the log files will contain the key-value pairs in their name, e.g. `./results/experiment3/alpha0.2beta1.0/`. Note the additional parameter `experiment`, which is set to the value `'grid'`.

The second option for running an experiment with parameter range combinations is a *list* experiment. Here, each experiment will contain values from both lists at the same indices. The number of sub-experiments created by this process is the length of the shortest list. In case of experiment 4 in Listing 8, `PyExperimentSuite` will create 3 experiments with the following parameters:

```
alpha = 0.0, beta = 0.5
alpha = 0.2, beta = 1.0
alpha = 0.3, beta = 2.0
```

The list for `alpha` would contain more values, but since there are not corresponding values in the `beta` list, the other experiments are discarded.

To control which type of experiment you want to conduct, you can define the special parameter `experiment` and assign `grid` or `list` to it (with or without quotation marks), as shown in Listing 8. The default value is `grid`. If you don’t want to create multiple experiments but use the assigned iterable object as value, use `single` as the `experiment` value.

Of course both mechanisms are not limited to two lists but can have arbitrary numbers of iterable objects. Just keep in mind that in the grid case, the number of experiments is the product of all list lengths, and in the list case, it is the length of the shortest list.

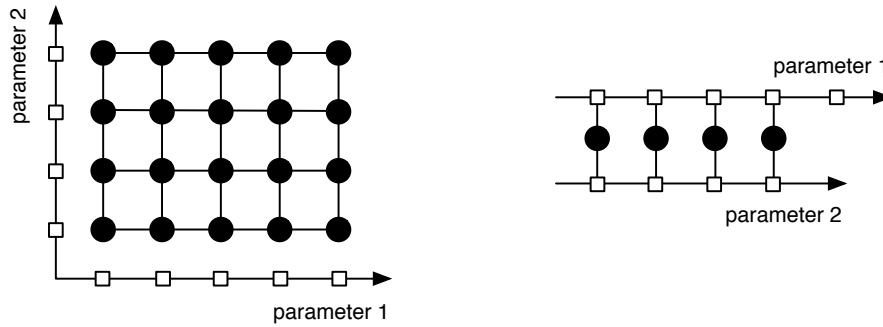


Figure 2: Two different types of experiments with parameter range combinations. The white squares represent discrete values of a parameter range, which is shown as an arrow. The black circles are experiments with a certain parameter combination. On the left side, a grid experiment is evaluated. On the right side, a list experiment is shown.

5 Run-Time Options

When calling your `suite.py` script from the command line, several options are available. The most important of them is the `-h` or `--help` flag. Calling the script with that option will present a list of all the available command line options. In this section, we will describe each of them and explain their usage. After reading this chapter, you are able to complete workflow step 6 of Section 3.1.

5.1 Help

```
$ python suite.py -h  
$ python suite.py --help
```

This option will not execute the script but instead show a list of available options for the suite.

5.2 Configuration File

```
$ python suite.py -c <config-file>
$ python suite.py --config=<config-file>
```

Specify your configuration file with the experiment definitions with this option. The default file is `experiments.cfg`.

5.3 Number of Cores

```
$ python suite.py -n <number>
$ python suite.py --numcores=<number>
```

This option lets you restrict PyExperimentSuite to a certain number of cores. If you specify this option, PyExperimentSuite will at most use the number of cores given. If this option is not present, PyExperimentSuite will use all the available cores it can find on the local machine.

When debugging your script, it is recommended to use only one core (with option `-n 1`). Details are explained in Section 7.4.

5.4 Deletion of Old Experiments

```
$ python suite.py -d
$ python suite.py --del
```

With this option you can tell PyExperimentSuite to delete all existing experiments before starting to parse the configuration file. This is useful if you made a mistake in the definition of your config file and need to restart the whole experiment set from scratch. This cannot be undone, so be careful when to use this option.

5.5 Selected Experiments

```
$ python suite.py -e <experiment>
$ python suite.py --experiment=<experiment>
```

If you don't want to execute all experiments defined in the config file but only one or a few selected ones, you can use this option. Specify each experiment you want to run explicitly with a separate `-e` option. If this option is not present, all defined experiments will be executed.

5.6 Browse Experiments

```
$ python suite.py -b
$ python suite.py --browse
$ python suite.py -B
$ python suite.py --Browse
```

The browse option does not execute the script (no experiments are started or continued) but returns a list of current experiments and additional information. The lower-case version of this option tells you each experiment's name, start and end time, the number of repetitions and iterations, and the progress of the experiment in percent, as shown below. The upper-case version of this option is more verbose and additionally tells you all the parameters for each experiment.

```
experiment ./results/experiment2/beta0.20alphano
  started 2010-07-28 00:10:22
  ended 2010-07-28 00:10:22
repetitions 1
iterations 10
progress 100%
```

5.7 Progress of Experiments

```
$ python suite.py -p
$ python suite.py --progress
```

Similar to the browse option, this progress option does not start or continue any experiments. It only shows a list of experiments and a progress bar of their completion. This is helpful if you have many experiments in the queue and quickly want to check how far the whole suite has progressed.

6 Result Retrieval

This section explains what you can do after your experiments have finished. It will explain where the log files are stored, what you will find in the log files, and how you can easily retrieve any relevant information within Python, without writing your own log file parser.

6.1 Log Files

The log files that PyExperimentSuite creates contain the key-value pairs that the `iterate()` function (ref. Section 3.3) returned as dictionary. Each line of a log file corresponds to one single iteration. They key-value pairs are stored in this format:

```
key1:value1 key2:value2 key3:value3 ...
```

For each repetition, a separate log file is created, and each log file is named `#.log` where `#` is the repetition number, starting with 0.

As the order of Python dictionary entries is unspecified, so is the order of the key-value pairs in the log file. The log file of our ongoing Neural Network example could look like this:

```
trainerror:0.32441 testerror:0.45531 n:0  
trainerror:0.29245 testerror:0.42648 n:1  
trainerror:0.27552 testerror:0.42796 n:2  
trainerror:0.23443 testerror:0.39441 n:3  
...
```

The log file would be stored at `./results/nn-learningrate/0.log` and there would be only one log file for the one repetition we requested.

If your experiment contained any parameter range evaluations with lists or other iterable objects (see Section 4.2 and 4.3), PyExperimentSuite will have created sub-directories under `./path/name/` with unique names based on the evaluated parameter combination. Within the sub-directories, the log files are stored as described before.

PyExperimentSuite not only stores the log files in the experiment folder, but one additional file: the configuration file of that single experiment. It is named `experiment.cfg` without the `s`, to distinguish it from the original `experiments.cfg` file. If you open

it, you will see that all the parameters that this particular experiment ran with, are there. There is no default section.

The existence of this additional configuration file has three reasons: it is a reminder for you, if you have forgotten which parameters exactly you evaluated in this experiment. PyExperimentSuite will also consult this file if you call the option `-B` or `--Browse` to tell you the value of all the parameters. Lastly, by specifying the `-c` or `--config` option, you can run only this particular experiment again, even if you have modified or deleted the original `experiments.cfg` file.

Experiments with parameter ranges create an intermediate `experiment.cfg` file in the `./path/name` folder and one for each `./path/name/sub-dir` folder as well.

The log files can be used to post-process your results, or plot or otherwise display them. In order to do this, you would usually have to write a parser, that can read the `key:value` syntax. PyExperimentSuite ships with a build-in parser already, that is accessible via a Python interface. The next section explains, how to retrieve your stored results.

6.2 Python Interface

PyExperimentSuite comes with several functions, that are intended to be used to retrieve data after the experiments have run. This section describes the final workflow step 7 from the list in Section 3.1.

All the functions intended to use for data retrieval start with the prefix `get_`. They are described in the following sections. Throughout the examples, we assume that you entered a Python console, imported your class `MySuite` from `suite.py` and created a `mysuite` object, like shown in Listing 9. For post-processing examples, we also import `numpy` and `matplotlib` functions. While `matplotlib` is incredibly useful to visualize your results, it is not a prerequisite for the main functionality of PyExperimentSuite but simply a part of the following examples.

Listing 9: Example to retrieve the results stored in log files.

```
>>> from suite import MySuite
>>> from numpy import *
>>> from matplotlib import pyplot as plt
>>> mysuite = MySuite()
```

6.3 Retrieving Information about Experiments

The first data retrieval function you might want to use is `get_exps()`. This function goes through all existing subdirectories and checks if they contain an `experiment.cfg` file, and are thus identified as experiments. It returns the full paths of all existing experiments (finished or not) under the current directory as a list.

The function has an optional parameter `path`, that specifies the directory of where to start with the search.

```
>>> mysuite.get_exps()
>>> mysuite.get_exps(path='.')
```

Another useful function is `get_exp()` (note the missing `s`).

```
>>> mysuite.get_exp(name, path='.')
```

This function takes the name of an experiment, as you put it in the config file section in square brackets, goes through all subdirectories and tries to locate the experiment. It then returns the full path of the found experiment. This function is useful if you know your experiment name but not the path anymore. It also has an optional parameter `path` to specify the directory of where to start the search.

All data retrieval functions, except for `get_exps()` and `get_exp()`, require an experiment identifier `exp` in their function call, which is the path and name of the experiment, e.g. `./results/experiment1`. The two above functions return this path, so you can use them for the functions that follow below.

The third function that gives you information about your experiments is `get_params()`.

```
>>> mysuite.get_params(exp)
```

This function takes the location of an experiment and returns all the parameters of the experiment, as listed in the `experiment.cfg` file. The return type is a dictionary.

6.4 Single Values

The most atomic data retrieval task is to access a single value of a single iteration. You could, for example, want to find the value of `testerror` after 25 iterations in the first repetition of experiment `nn-learningrate`. The function `get_value()` does exactly that.

```
>>> mysuite.get_value(exp, rep, tags, which)
```

The function takes four parameters: `exp` is the directory of the experiment you would like to access, `rep` specifies the repetition number. `tags` can be a string or a list of strings. If it is a string, it needs to contain the key which you would like to access. This is the key you added to the dictionary in method `iterate()`. If you want to access several keys at once, pass a list of strings to the function, and if you want to retrieve all keys, simply pass the string `'all'`. Finally, you need to specify, `which` value(s) you would like to access. The choices are the strings `'last'`, `'min'`, `'max'` or an integer. `'last'` will return the value from the very last iteration. `'min'` and `'max'` return the minimal or maximal value over all iterations. If you pass an integer to the `which` argument, you will get the value at this specific iteration. For the afore-mentioned example, the call would be

```
>>> mysuite.get_value('./results/nn-learningrate', 0,
                    'testerror', 25)
```

If you requested only one key, the return value will be a scalar. If you requested several or all keys, you will get a dictionary with key-value pairs as a result.

6.5 Single Values from Parameter Range Experiments

Retrieving a single value from a sub-experiment created by parameter ranges works the same, except the experiment location is a bit longer. Use the full experiment path `./path/name/sub-exp` to retrieve single values.

There is, however, a second function, that is made to work with parameter ranges, and it is particular useful with grid experiments. If you have tested a grid of several parameters, and you would like to retrieve values along one particular parameter axis and fixing the other ranges, the function `get_values_fix_params()` might be of use to you.

```
>>> mysuite.get_values_fix_params(exp, rep, tags, which, **kwargs)
```

It uses the same syntax as `get_value()` but has an additional `**kwargs` argument, which means you can pass an arbitrary number of additional keyword-value pairs in the function. The keywords expected here are the keys you would like to keep fixed to a certain value. Basically, for each keyword you add, you slice the (possibly multi-dimensional) grid along one axis and only consider experiments, that lie on that axis. Refer to the left side of Figure 2 again for visual support.

Let's have a look at experiment 3 from Listing 8, which defined two parameter ranges in a grid fashion:

```
alpha = arange(0.0, 1.0, 0.2)
beta = [0.5, 1.0, 2.0]
```

This definition will create 15 experiments (because `alpha` evaluates to `[0.0, 0.2, 0.4, 0.6, 0.8]`). If this was part of our Neural Network example, and we were looking for the smallest test error under the condition that `alpha` was 0.2, this is how we would call the function:

```
>>> values, params = mysuite.get_values_fix_params(
    './results/experiment3', 0, 'testerror', 'min', alpha=0.2)
```

As you can see in the function call, this function actually returns a tuple of two different things: the actual values, and the experiment parameters. Both return values are lists of equal lengths. The first one contains results in the form that you would expect from `get_value()`: either scalars or dictionaries of key-value pairs. The second return value contains equally many dictionaries: each dictionary consists of all the parameters of the experiments that matched the criteria given by `**kwargs`.

You can add as few or many conditions to the function as you like, not just one. Each condition limits the returned experiments and values further. If you call the function without any conditions, it will go over all sub-experiments. As an example, if you evaluated several different parameter ranges in a grid and just want to find the overall lowest test error, call this function without any `**kwargs` arguments.

I recommend that you create a small toy grid experiment with `PyExperimentSuite` and call this function with different `**kwargs` parameters, to get a grasp of how this function exactly works.

6.6 Histories

Another common task with `PyExperimentSuite` is to retrieve a whole history over all iterations of a certain key, for example if you want to plot how the error of an optimization problem slowly decreased with each iteration. The function `get_history()` can help you with this.

```
>>> mysuite.get_history(exp, rep, tags)
```

This function expects the location of your experiment, the index of the repetition you would like to access, and the keys you want to retrieve, either a single string, a list of strings, or the string 'all'.

The function returns a list of all values of that key, or a dictionary of lists with the corresponding keys.

In case you want to plot the history with the separate module matplotlib, you could call:

```
>>> plt.plot(mysuite.get_history('./results/experiment1', 0,
                                'testerror'), '-o', label='test error')
```

6.7 Histories from Parameter Range Experiments

Just as with the `get_value()` method described above, histories can also be returned from parameter range experiments. If you know the exact sub-experiment already, you can pass the full path to the `get_history()` function and will receive the history.

If you want to retrieve several histories over a set of parameter range experiments, the function `get_histories_fix_params()` is what you need.

```
>>> histories, params = mysuite.get_histories_fix_params(
                                exp, rep, tags, **kwargs)
```

It works exactly like the single value version `get_values_fix_params()` but returns history lists instead of single values.

6.8 Aggregated Histories over Repetitions

There is another useful history-retrieving function, which is most useful if you executed many repetitions of the same experiment. One reason to do this, is because your experiment might be of stochastic nature and you want to average over all these repetitions.

```
>>> mysuite.get_histories_over_repetitions(exp, tags, aggregate)
```

This function requires the location of your experiment, the key(s) you are interested in (again as single string, list of string or the string 'all'), and an aggregation function. Typical aggregation functions could be `sum`, `mean`, `max`, etc. The function will not return the history of one single repetition, but use all repetitions and maps the values of each repetition to the aggregation function. Let's say the aggregation function is called `aggr()` and the value at repetition r and iteration i is defined as v_i^r . Repetitions range from 0 to R and iterations range from 0 to I . The result is then a new list, that contains values with same iteration indices, to which `aggr()` has been applied:

$$[\text{aggr}(v_0^0, v_0^1, \dots, v_0^R), \text{aggr}(v_1^0, v_1^1, \dots, v_1^R), \dots, \text{aggr}(v_I^0, v_I^1, \dots, v_I^R)]$$

One common use of this function is to average histories over all repetitions with the aggregation function `mean` (imported from `numpy`):

```
>>> mysuite.get_histories_over_repetitions(
    './results/experiment1', 'testerror', mean)
```

7 Examples and Best Practices

This section contains some more examples on how to use `PyExperimentSuite` and demonstrates some less common cases that you might be confronted with.

7.1 Strings in Iterable Objects

In Section 4 you learned about the configuration file and the syntax of defining parameters:

```
parameter = value
```

You also read that strings don't need to be surrounded by quotation marks, because everything that cannot be evaluated to a pythonic integer, float or object will be interpreted as strings. If you want to use parameter ranges over strings, however, the quotation marks become necessary again:

```
parameter = ['string1', 'string2', 'string3']
```

Why is this the case? `PyExperimentSuite` will again try to evaluate the value of the parameter—as a whole. It would stumble over the unquoted literals, assuming they were undefined variables, and conclude that it can't evaluate the list. The result

would be a parameter that interprets the whole definition of the list as a single string, including the square brackets around it.

7.2 Passing Classes as Parameters

A common use case of `PyExperimentSuite` is to test different algorithms or classes against each other. One possibility is of course to define a flag, that your `reset()` method uses to distinguish between the different program alternatives.

Let's say you have this line in your config file:

```
newalgorithm = [0, 1]
```

`PyExperimentSuite` will execute two experiments, once with `newalgorithm` set to 0 and once where it is set to 1. Your `reset()` method could then look like Listing 10.

Listing 10: Comparing different classes against one another with if-statement.

```
def reset(self, params, reps):
    if params['newalgorithm']:
        # initialize new algorithm
        self.algorithm = NewAlgorithmClass()
    else:
        # initialize old algorithm
        self.algorithm = OldAlgorithmClass()
```

Another, perhaps more elegant way is to use Python's `eval()` function⁴ to test different classes. Define the parameter in the config file as a list of strings (but do use quotation marks, as explained in Section 7.1):

```
algorithm = ['NewAlgorithmClass', 'OldAlgorithmClass']
```

Now you can write a much shorter `reset()` method (Listing 11). Note the additional `+ '()'` in the `eval()` statement, which is necessary to create a new object from the class instead of assigning the class itself to `self.algorithm`. If you have additional parameters to pass to the algorithm class upon initialization, you can do that as well within the `eval()` statement with string concatenation.

⁴Although you often read that `eval()` is evil and that you shouldn't use it, I find it perfectly suited for this task and don't see the disadvantage of using it here. But the decision is up to you.

Listing 11: Comparing different classes against one another with eval-statement.

```
def reset(self, params, reps):  
    # convert string to class  
    self.algorithm = eval(params['algorithm'] + '()')
```

7.3 Passing Lists as Parameters

While the parameter range mechanism with iterable objects is convenient, sometimes you might want to pass an actual list as a single parameter without evaluating it as a range. You can tell PyExperimentSuite that your lists should not be transformed into parameter range experiments, by defining the parameter `experiment` in your config file and assigning it the string `'single'`.

```
experiment = 'single'  
alpha = [0.5, 1.0, 1.5]
```

Now the list is assigned to the parameter as a whole and no sub-experiments are created. In the above case, you could access `params['alpha']` within your `reset()` or `iterate()` methods and would get a list containing the three values 0.5, 1.0, 1.5.

It is currently not possible to mix grid, list and single experiments, i.e. one list will be returned as is where the other lists form a grid.

7.4 Debugging PyExperimentSuite Scripts

If you request more than one core to execute the experiments, PyExperimentSuite will use the multiprocessing package and create a worker pool for the required number of processes. If you specifically request to use only 1 core (with the runtime option `-n 1`), PyExperimentSuite will not go through the multiprocessing package but run the experiment directly from the main process. This is especially useful if you are debugging your script, because you will get the exact location in the code where an exception occurred. Using multiple processes, this information is not available and you will have trouble finding any bugs.