Python Experiment Suite Implementation

Thomas Rückstieß
Technische Universität München, Germany
ruecksti@in.tum.de

Jürgen Schmidhuber
Dalle Molle Institute for Artificial Intelligence, Lugano, Switzerland
juergen@idsia.ch

Abstract

This manuscript describes the implementation of a generic experiment management tool called Python Experiment Suite, an open source software framework written in Python, that supports scientists, engineers and others to conduct automated software experiments on a larger scale with features like parameter evaluations in grid search manner, result logging and support for multiple cores, amongst others.

1. Introduction

The Python Experiment Suite (Rückstieß and Schmidhuber, 2011) is a software framework that supports scientists, engineers, students and others in running software experiments and evaluations for their research and work. It provides a wrapper around any Python-based experiment and contains many useful features: parameter ranges and combinations can be evaluated automatically, where different experiment architectures (e.g. grid search) are available. The suite also takes care of logging results into files, can handle experiment interruption and continuation, for instance after process termination by power failure, supports execution on multiple cores and contains a convenient Python interface to retrieve the stored results. The Python Experiment Suite also uses configuration files for parameter definititions and setup of complex experiments without the need to modify code.

The software is released under the modified BSD License and is available for download at: `http://github.com/rueckstiess/expsuite`. It requires Python Version 2.6 (because of its dependency on the `multiprocessing` package) and the numerical computation package `NumPy`. Other than that, the Python Experiment Suite is a stand-alone module.

The original publication (Rückstieß and Schmidhuber, 2011) is directed to audience who wants to use the software as is for their experiments. It explains how to set up and conduct experiments, how to use the configuration file and finally how to retrieve the results.

This manuscript provides additional information on the implementation of the Python Experiment Suite including the source code. It is aimed towards developers, who want to extend and modify the software to better match their needs. Section 1.1 therefore gives more details on execution and program flow and explains some of the more complex parts of the software.

Section 2 presents the full source code for the Python Experiment Suite. Finally, Section 3 contains the source code of two examples on how to use the Python Experiment Suite. These examples are also included in the distribution under the `/examples` subdirectory.

1.1 Experiment Execution Details

The Python Experiment Suite is developed as an object-oriented new-style Python class[1], from which the users should derive their experiment suite (see the examples in Section 3). The class offers two methods that need to be overwritten in the subclass: `reset()` and `iterate()`. The `reset()` method takes care of experiment initialization, data loading and other preparation steps, while the `iterate()` method executes a single step in the experiment progress. The number of calls to `iterate()` can be set in the configuration file with the parameter `iterations`.

Other method stubs can be implemented as well, e.g. `save_state()` and `restore_state()`. These methods are required for the optional restore functionality, described in (Rückstieß and Schmidhuber, 2011).

The `start()` method initiates the experiment process. It will parse the command line parameters, read the main configuration file and then start the experiments by calling the `do_experiments()` method with the parameters list. This method checks whether several repetitions of the same experiment are requested, extends the parameters list appropriately and then creates the multiprocessing pool and spawns several processes, depending on the number of available cores. If only one core is available (or the number has been restricted to one via the command line option `–n1`), the experiment will be executed directly in the main process, instead of creating a worker pool. For debugging your code it is recommended to always use only one core, as the error messages will point to the correct line in the code, where the error occurred. When using multiple cores, the error messages will only contain the line number of where the process was started, which makes tracing back any errors much harder.

The processes for the worker pool are created with the `Pool.map()` method, which works like the built-in Python equivalent `map()` but on multiple processes. Note, that the function `mp_runrep()`, which is passed to `map()`, is a global function, not defined in the `PyExperimentSuite` class. Trying to map a class method directly will cause an exception in the `multiprocessing` package. The work-around is to use a global function outside the class, unpack the `self` from the arguments and call the method from there. The first argument in the `args` list is the object (represented by `self`), followed by the original arguments for the `run_rep()` method. The method runs a single repetition of a certain experiment (defined by the parameters `params`) by first calling the `reset()` method and then repeatedly the `iterate()` method. The key-value pairs from the dictionary returned by `iterate()` will be used to populate the log files. The `run_rep()` method also checks, if results for this repetition already exist, and continues from where it left off if the `restore_supported` flag is set to `True`. Otherwise, it will delete existing iterations and start the current repetition again from iteration 0. The `reset()` and `iterate()` method calls receive the current experiment parameters `params` and the repetition number `rep`, and `iterate()` additionally receives the current iteration number `n`.

[1] New-style classes in Python derive from the `object` class and extend the standard classes by many useful features, see `http://docs.python.org/reference/datamodel.html#newstyle` for more information.

2. The Python Experiment Suite Source Code

```
##############################################################################
#
# PyExperimentSuite
#
# Derive your experiment from the PyExperimentSuite, fill in the reset() and
# iterate() methods, and define your defaults and experiments variables
# in a config file.
# PyExperimentSuite will create directories, run the experiments and store the
# logged data. An aborted experiment can be resumed at any time. If you want
# to resume it on iteration level (instead of repetition level) you need to
# implement the restore_state and save_state method and make sure the
# restore_supported variable is set to True.
#
# For more information, consult the included documentation.pdf file.
#
# Licensed under the modified BSD License. See LICENSE file in same folder.
#
# Copyright 2010 - Thomas Rueckstiess
#
##############################################################################

from ConfigParser import ConfigParser
from multiprocessing import Process, Pool, cpu_count
from numpy import *
import os
import sys
import time
import itertools
import re
import optparse
import types


def mp_runrep(args):
    """ Helper function to allow multiprocessing support. """
    return PyExperimentSuite.run_rep(*args)


def progress(params, rep):
    """ Helper function to calculate the progress made on one experiment. """
    name = params['name']
    fullpath = os.path.join(params['path'], params['name'])
    logname = os.path.join(fullpath, '%i.log'%rep)
    if os.path.exists(logname):
        logfile = open(logname, 'r')
        lines = logfile.readlines()
        logfile.close()
        return int(100 * len(lines) / params['iterations'])
    else:
        return 0
```

```
def convert_param_to_dirname(param):
    """ Helper function to convert a parameter value to a valid directory
        name. """
    if type(param) == types.StringType:
        return param
    else:
        return re.sub("0+$", '0', '%f'%param)


class PyExperimentSuite(object):

    # change this in subclass, if you support restoring on iteration level
    restore_supported = False

    def __init__(self):
        self.parse_opt()
        self.parse_cfg()

        # list of keys, that had to be renamed because they contained spaces
        self.key_warning_issued = []

    def parse_opt(self):
        """ parses the command line options for different settings. """
        optparser = optparse.OptionParser()
        optparser.add_option('-c', '--config',
            action='store', dest='config', type='string',
            default='experiments.cfg', help="your experiments config file")
        optparser.add_option('-n', '--numcores',
            action='store', dest='ncores', type='int', default=cpu_count(),
            help="number of processes you want to use, " +
                 "default is %i"%cpu_count())
        optparser.add_option('-d', '--del',
            action='store_true', dest='delete', default=False,
            help="delete experiment folder if it exists")
        optparser.add_option('-e', '--experiment',
            action='append', dest='experiments', type='string',
            help="run only selected experiments, by default run all" +
                  "experiments in config file.")
        optparser.add_option('-b', '--browse',
            action='store_true', dest='browse', default=False,
            help="browse existing experiments.")
        optparser.add_option('-B', '--Browse',
            action='store_true', dest='browse_big', default=False,
            help="browse existing experiments, more verbose than -b")
        optparser.add_option('-p', '--progress',
            action='store_true', dest='progress', default=False,
            help="like browse, but only shows name and progress bar")

        options, args = optparser.parse_args()
        self.options = options
        return options, args

    def parse_cfg(self):
        """ parses the given config file for experiments. """
        self.cfgparser = ConfigParser()
        if not self.cfgparser.read(self.options.config):
            raise SystemExit('config file %s not found.'%self.options.config)

    def mkdir(self, path):
```

```
    """ create a directory if it does not exist. """
    if not os.path.exists(path):
        os.makedirs(path)

def get_exps(self, path='.'):
    """ go through all subdirectories starting at path and return the
        experiment identifiers (= directory names) of all existing
        experiments. A directory is considered an experiment if it
        contains a experiment.cfg file.
    """
    exps = []
    for dp, dn, fn in os.walk(path):
        if 'experiment.cfg' in fn:
            subdirs = [os.path.join(dp, d) for d in os.listdir(dp) if
                os.path.isdir(os.path.join(dp, d))]
            if all(map(lambda s: self.get_exps(s) == [], subdirs)):
                exps.append(dp)
    return exps

def items_to_params(self, items):
    """ evaluate the found items (strings) to become floats, ints
        or lists. """
    params = {}
    for t, v in items:
        try:
            # try to evaluate parameter (float, int, list)
            if v in ['grid', 'list']:
                params[t] = v
            else:
                params[t] = eval(v)
            if isinstance(params[t], ndarray):
                params[t] = params[t].tolist()
        except (NameError, SyntaxError):
            # otherwise assume string
            params[t] = v
    return params

def get_params(self, exp, cfgname='experiment.cfg'):
    """ reads the parameters of the experiment (= path) given.
    """
    cfgp = ConfigParser()
    cfgp.read(os.path.join(exp, cfgname))
    section = cfgp.sections()[0]
    params = self.items_to_params(cfgp.items(section))
    params['name'] = section
    return params

def get_exp(self, name, path='.'):
    """ given an experiment name (used in section titles), this function
        returns the correct path of the experiment.
    """
    exps = []
    for dp, dn, df in os.walk(path):
        if 'experiment.cfg' in df:
            cfgp = ConfigParser()
            cfgp.read(os.path.join(dp, 'experiment.cfg'))
            if name in cfgp.sections():
                exps.append(dp)
    return exps
```

```
def write_config_file(self, params, path):
    """ write a config file for this single exp in the folder path.
    """
    cfgp = ConfigParser()
    cfgp.add_section(params['name'])
    for p in params:
        if p == 'name':
            continue
        cfgp.set(params['name'], p, params[p])
    f = open(os.path.join(path, 'experiment.cfg'), 'w')
    cfgp.write(f)
    f.close()

def get_history(self, exp, rep, tags):
    """ returns the whole history for one experiment and one repetition.
        tags can be a string or a list of strings. if tags is a string,
        the history is returned as list of values, if tags is a list of
        strings or 'all', history is returned as a dictionary of lists
        of values.
    """
    params = self.get_params(exp)

    if params == None:
        raise SystemExit('experiment %s not found.'%exp)

    # make list of tags, even if it is only one
    if tags != 'all' and not hasattr(tags, '__iter__'):
        tags = [tags]

    results = {}
    logfile = os.path.join(exp, '%i.log'%rep)
    try:
        f = open(logfile)
    except IOError:
        if len(tags) == 1:
            return []
        else:
            return {}

    for line in f:
        pairs = line.split()
        for pair in pairs:
            tag, val = pair.split(':')
            if tags == 'all' or tag in tags:
                if not tag in results:
                    try:
                        results[tag] = [eval(val)]
                    except (NameError, SyntaxError):
                        results[tag] = [val]
                else:
                    try:
                        results[tag].append(eval(val))
                    except (NameError, SyntaxError):
                        results[tag].append(val)

    f.close()
    if len(results) == 0:
        if len(tags) == 1:
```

```
            return []
        else:
            return {}
        # raise ValueError('tag(s) not found: %s'%str(tags))
    if len(tags) == 1:
        return results[results.keys()[0]]
    else:
        return results

def get_value(self, exp, rep, tags, which='last'):
    """ Like get_history(..) but returns only one single value rather
        than the whole list.
        tags can be a string or a list of strings. if tags is a string,
        the history is returned as a single value, if tags is a list of
        strings, history is returned as a dictionary of values.
        'which' can be one of the following:
            last: returns the last value of the history
             min: returns the minimum value of the history
             max: returns the maximum value of the history
               #: (int) returns the value at that index
    """
    history = self.get_history(exp, rep, tags)

    # empty histories always return None
    if len(history) == 0:
        return None

    # distinguish dictionary (several tags) from list
    if type(history) == dict:
        for h in history:
            if which == 'last':
                history[h] = history[h][-1]
            if which == 'min':
                history[h] = min(history[h])
            if which == 'max':
                history[h] = max(history[h])
            if type(which) == int:
                history[h] = history[h][which]
        return history

    else:
        if which == 'last':
            return history[-1]
        if which == 'min':
            return min(history)
        if which == 'max':
            return max(history)
        if type(which) == int:
            return history[which]
        else:
            return None

def get_values_fix_params(self, exp, rep, tag, which='last', **kwargs):
    """ this function uses get_value(..) but returns all values where the
        subexperiments match the additional kwargs arguments. if
        alpha=1.0, beta=0.01 is given, then only those experiment values
        are returned, as a list.
    """
    subexps = self.get_exps(exp)
```

```
    tagvalues = ['%s%s'%(k, convert_param_to_dirname(kwargs[k])) for
        k in kwargs]

    values = [self.get_value(se, rep, tag, which) for se in subexps if
        all(map(lambda tv: tv in se, tagvalues))]
    params = [self.get_params(se) for se in subexps if
        all(map(lambda tv: tv in se, tagvalues))]

    return values, params

def get_histories_fix_params(self, exp, rep, tag, **kwargs):
    """ this function uses get_history(..) but returns all histories
        where the subexperiments match the additional kwargs arguments.
        if alpha=1.0, beta = 0.01 is given, then only those experiment
        histories are returned, as a list.
    """
    subexps = self.get_exps(exp)
    tagvalues = [re.sub("0+$", '0', '%s%f'%(k, kwargs[k])) for k in
        kwargs]

    histories = [self.get_history(se, rep, tag) for se in subexps if \
        all(map(lambda tv: tv in se, tagvalues))]
    params = [self.get_params(se) for se in subexps if \
        all(map(lambda tv: tv in se, tagvalues))]

    return histories, params

def get_histories_over_repetitions(self, exp, tags, aggregate):
    """ this function gets all histories of all repetitions using
        get_history() on the given tag(s), and then applies the
        function given by 'aggregate' to all corresponding values
        in each history over all iterations. Typical aggregate
        functions could be 'mean' or 'max'.
    """
    params = self.get_params(exp)

    # explicitly make tags list in case of 'all'
    if tags == 'all':
        tags = self.get_history(exp, 0, 'all').keys()

    # make list of tags if it is just a string
    if not hasattr(tags, '__iter__'):
        tags = [tags]

    results = {}
    for tag in tags:
        # get all histories
        histories = zeros((params['repetitions'], params['iterations']))
        skipped = []
        for i in range(params['repetitions']):
            try:
                histories[i, :] = self.get_history(exp, i, tag)
            except ValueError:
                h = self.get_history(exp, i, tag)
                if len(h) == 0:
                    # history not existent, skip it
                    print('warning: history %i has length 0 (expected: '+
                        '%i). it will be skipped.'%(i, \
                        params['iterations']))
```

```
                    skipped.append(i)
                elif len(h) > params['iterations']:
                    # if history too long, crop it
                    print('warning: history %i has length %i (expected: '+
                        '%i). it will be truncated.'%(i, len(h), \
                        params['iterations']))
                    h = h[:params['iterations']]
                    histories[i, :] = h
                elif len(h) < params['iterations']:
                    # if history too short, crop everything else
                    print('warning: history %i has length %i (expected: '+
                        '%i). all other histories will be truncated.'%(i,\
                        len(h), params['iterations']))
                    params['iterations'] = len(h)
                    histories = histories[:, :params['iterations']]

            # remove all rows that have been skipped
            histories = delete(histories, skipped, axis=0)
            params['repetitions'] -= len(skipped)

            # calculate result from each column with aggregation function
            aggregated = zeros(params['iterations'])
            for i in range(params['iterations']):
                aggregated[i] = aggregate(histories[:, i])

            # if only one tag is requested, return list immediately, otherwise
            # append to dictionary
            if len(tags) == 1:
                return aggregated
            else:
                results[tag] = aggregated

        return results

    def browse(self):
        """ go through all subfolders (starting at '.') and return information
            about the existing experiments. if the -B option is given, all
            parameters are shown, -b only displays the most important ones.
            this function does *not* execute any experiments.
        """
        for d in self.get_exps('.'):
            params = self.get_params(d)
            name = params['name']
            basename = name.split('/')[0]
            # if -e option is used, only show requested experiments
            if self.options.experiments and basename not in \
                self.options.experiments:
                continue

            fullpath = os.path.join(params['path'], name)

            # calculate progress
            prog = 0
            for i in range(params['repetitions']):
                prog += progress(params, i)
            prog /= params['repetitions']

            # if progress flag is set, only show the progress bars
            if self.options.progress:
```

```
            bar = "["
            bar += "="*int(prog/4)
            bar += " "*int(25-prog/4)
            bar += "]"
            print '%3i%% %27s %s'%(prog, bar, d)
            continue

        print '%16s %s'%('experiment', d)

        try:
            minfile = min(
                (os.path.join(dirname, filename)
                for dirname, dirnames, filenames in os.walk(fullpath)
                for filename in filenames
                if filename.endswith(('.log', '.cfg'))),
                key=lambda fn: os.stat(fn).st_mtime)

            maxfile = max(
                (os.path.join(dirname, filename)
                for dirname, dirnames, filenames in os.walk(fullpath)
                for filename in filenames
                if filename.endswith(('.log', '.cfg'))),
                key=lambda fn: os.stat(fn).st_mtime)
        except ValueError:
            print '         started %s'%'not yet'

        else:
            print '         started %s'%time.strftime('%Y-%m-%d %H:%M:%S',
                time.localtime(os.stat(minfile).st_mtime))
            print '           ended %s'%time.strftime('%Y-%m-%d %H:%M:%S',
                time.localtime(os.stat(maxfile).st_mtime))

        for k in ['repetitions', 'iterations']:
            print '%16s %s'%(k, params[k])

        print '%16s %i%%'%('progress', prog)

        if self.options.browse_big:
            # more verbose output
            for p in [p for p in params if p not in ('repetitions', \
                'iterations', 'path', 'name')]:
                print '%16s %s'%(p, params[p])

        print

    def expand_param_list(self, paramlist):
        """ expands the parameters list according to one of these schemes:
            grid: every list item is combined with every other list item
            list: every n-th list item of parameter lists are combined
        """
        # for one single experiment, still wrap it in list
        if type(paramlist) == types.DictType:
            paramlist = [paramlist]

        # get all options that are iteratable and build all combinations
        # (grid) or tuples (list)
        iparamlist = []
        for params in paramlist:
            if ('experiment' in params and params['experiment'] == 'single'):
```

```
            iparamlist.append(params)
        else:
            iterparams = [p for p in params if \
                hasattr(params[p], '__iter__')]
            if len(iterparams) > 0:
                # write intermediate config file
                self.mkdir(os.path.join(params['path'], params['name']))
                self.write_config_file(params,
                    os.path.join(params['path'], params['name']))

                # create sub experiments (check if grid/list is requested)
                if 'experiment' in params and \
                    params['experiment'] == 'list':
                    iterfunc = itertools.izip
                elif ('experiment' not in params) or ('experiment' in \
                    params and params['experiment'] == 'grid'):
                    iterfunc = itertools.product
                else:
                    raise SystemExit("unexpected value '%s' for para" + \
                    "meter 'experiment'. Use 'grid', 'list' or " + \
                    "'single'."%params['experiment'])

                for il in iterfunc(*[params[p] for p in iterparams]):
                    par = params.copy()
                    converted = str(zip(iterparams,
                        map(convert_param_to_dirname, il)))
                    par['name'] = par['name'] + '/' + \
                        re.sub("[' \[\],()]", '', converted)
                    for i, ip in enumerate(iterparams):
                        par[ip] = il[i]
                    iparamlist.append(par)
            else:
                iparamlist.append(params)

    return iparamlist

def create_dir(self, params, delete=False):
    """ creates a subdirectory for the experiment, and deletes existing
        files, if the delete flag is true. then writes the current
        experiment.cfg file in the folder.
    """
    # create experiment path and subdir
    fullpath = os.path.join(params['path'], params['name'])
    self.mkdir(fullpath)

    # delete old histories if --del flag is active
    if delete:
        os.system('rm %s/*' % fullpath)

    # write a config file for this single exp. in the folder
    self.write_config_file(params, fullpath)

def start(self):
    """ starts the experiments as given in the config file. """

    # if -b, -B or -p option is set, only show information, don't
    # start the experiments
    if self.options.browse or self.options.browse_big or \
        self.options.progress:
```

```
            self.browse()
            raise SystemExit

        # read main configuration file
        paramlist = []
        for exp in self.cfgparser.sections():
            if not self.options.experiments or exp in
                   self.options.experiments:
                params = self.items_to_params(self.cfgparser.items(exp))
                params['name'] = exp
                paramlist.append(params)

        self.do_experiment(paramlist)

    def do_experiment(self, params):
        """ runs one experiment programatically and returns.
            params: either parameter dictionary (for one single experiment)
            or a list of parameter dictionaries (for several experiments).
        """
        paramlist = self.expand_param_list(params)

        # create directories, write config files
        for pl in paramlist:
            # check for required param keys
            if ('name' in pl) and ('iterations' in pl) and \
                ('repetitions' in pl) and ('path' in pl):
                self.create_dir(pl, self.options.delete)
            else:
                print 'Error: parameter set does not contain all required ' +\
                    'keys: name, iterations, repetitions, path'
                return False

        # create experiment list
        explist = []

        # expand paramlist for all repetitions and add self and rep number
        for p in paramlist:
            explist.extend(zip([self]*p['repetitions'], \
                [p]*p['repetitions'], xrange(p['repetitions'])))

        # if only 1 process is required call each experiment seperately
        # (no worker pool)
        if self.options.ncores == 1:
            for e in explist:
                mp_runrep(e)
        else:
            # create worker processes
            pool = Pool(processes=self.options.ncores)
            pool.map(mp_runrep, explist)

        return True

    def run_rep(self, params, rep):
        """ run a single repetition including directory creation, log
            files, etc.
        """
        name = params['name']
        fullpath = os.path.join(params['path'], params['name'])
        logname = os.path.join(fullpath, '%i.log'%rep)
```

```
    # check if repetition exists and has been completed
    restore = 0
    if os.path.exists(logname):
        logfile = open(logname, 'r')
        lines = logfile.readlines()
        logfile.close()

        # if completed, continue loop
        if 'iterations' in params and len(lines) == params['iterations']:
            return False
        # if not completed, check if restore_state is supported
        if not self.restore_supported:
            # not supported, delete repetition and start over
            # print 'restore not supported, deleting %s' % logname
            os.remove(logname)
            restore = 0
        else:
            restore = len(lines)

    self.reset(params, rep)

    if restore:
        logfile = open(logname, 'a')
        self.restore_state(params, rep, restore)
    else:
        logfile = open(logname, 'w')

    # loop through iterations and call iterate
    for it in xrange(restore, params['iterations']):
        dic = self.iterate(params, rep, it)
        if self.restore_supported:
            self.save_state(params, rep, it)

        # replace all spaces in keys with underscores
        for k in dic:
            if ' ' in k:
                newk = k.replace(' ', '_')
                dic[newk] = dic[k]
                del dic[k]
                # issue warning but only once per key
                if k not in self.key_warning_issued:
                    print "warning: key '%s' contained spaces and was "+ \
                        "renamed to '%s'"%(k, newk)
                    self.key_warning_issued.append(k)

        # build string from dictionary
        outstr = ' '.join(map(lambda x: '%s:%s'%(x[0], str(x[1])),
            dic.items()))
        logfile.write(outstr + '\n')
        logfile.flush()
    logfile.close()

def reset(self, params, rep):
    """ needs to be implemented by subclass. """
    pass

def iterate(self, params, rep, n):
    """ needs to be implemented by subclass. """
    ret = {'iteration': n, 'repetition': rep}
```

```
        return ret

    def save_state(self, params, rep, n):
        """ optionally can be implemented by subclass. """
        pass

    def restore_state(self, params, rep, n):
        """ if the experiment supports restarting within a repetition
            (on iteration level), load necessary stored state in this
            function. Otherwise, restarting will be done on repetition
            level, deleting all unfinished repetitions and restarting
            the experiments.
        """
        pass
```

3. Usage Examples for the Python Experiment Suite

In this section we present two examples on how to use the Python Experiment Suite. Each Example defines a class that derives from the PyExperimentSuite class, and implements a few required class methods, in particular the reset() and iterate() methods. While the "basic" example from Section 3.1 demonstrates the basic functionality of the Python Experiment Suite, the "random" example in Section 3.2 shows some more advanced features, like continuation after aborted execution. More examples are available in the distribution of the Python Experiment Suite under the `/examples` subfolder.

3.1 The "Basic" Example

This example shows how to derive a class from the PyExperimentSuite class, and how to implement the two required methods `reset()` and `iterate()`. In the former method, everything required for the experiment is set up and initialized, while the latter executes one calculation step. In this example though, the experiment does nothing but load 2 parameters alpha and beta from a config file, and return the current repetition and iteration number, together with the values of the two parameters for logging. Create a file called "suite.py" and add the following lines:

```
from expsuite import PyExperimentSuite


class MySuite(PyExperimentSuite):

    def reset(self, params, rep):
        """ for this basic example, nothing needs to be loaded or
            initialized."""
        pass

    def iterate(self, params, rep, n):
        """ this function does nothing but access the two parameters alpha
            and beta from the config file experiments.cfg and return them for
            the log files, together with the current repetition and iteration
            number.
        """
```

```
        # access the two config file parameters alpha and beta
        alpha = params['alpha']
        beta = params['beta']

        # return current repetition and iteration number and the 2 parameters
        ret = {'rep': rep, 'iter': n, 'alpha': alpha, 'beta': beta}
        return ret

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

The configuration file also needs to be present. Create a second file called “experiments.cfg” and place it in the same folder, containing these lines:

```
[DEFAULT]
repetitions = 5
iterations = 10
path = results

[myexperiment]
alpha = 1
beta = 0.1
```

Now run the experiment from the console, by executing `python suite.py`. The script creates a subfolder in the current directory called “results” and another folder in it with the experiment name “myexperiment”. In this folder, 4 files are created called “#.log” where # is the repetition number ranging from 0 to 4. Each of the files contains 10 lines, one for each iteration. The lines will all look similar to this:

```
alpha:1 rep:0 beta:0.1 iter:0
```

Instead of accessing the data directly through the log files, a Python API is also provided with the Suite. Open a Python shell, or create another Python script and run the following commands to retreive the logged data programmatically:

```
from suite import MySuite

# create the object of your suite
mysuite = MySuite()

# list of all experiments with get_exps()
exps = mysuite.get_exps()
print "list of available experiments:", exps

print "last value of 'beta' in repetition number 2 (2.log):", \
    mysuite.get_value(exps[0], 2, 'beta', 'last')

print "biggest value of 'iter' in repetition number 0 (0.log):", \
    mysuite.get_value(exps[0], 0, 'iter', 'max')

print "history of values of 'iter' in repetition number 4 (4.log):", \
    mysuite.get_history(exps[0], 4, 'iter')
```

3.2 The "Random" Example

This example demonstrates a slightly more complex use case of the Python Experiment Suite, including the optional *restore* functionality, where an experiment can be interrupted at any time and continues exactly where it left off.

This particular experiment verifies a simple stochastic law, namely that the sample mean of random numbers drawn from a normal distribution does converge to the actual mean of the distribution itself.

In the `reset()` method, the array that will contain the random numbers is initialized with zeros, and the random number generator is seeded with the given seed from the config file parameter `seed`.

The `iterate()` method then draws a normally distributed random number with mean and standard deviation from config file parameters `mean` and `std`. It then calculates the sample mean of all drawn numbers so far, calculates the offset to the real mean and returns this offset together with some other information for logging purposes. The code below should be placed in a file called "suite.py".

```
from expsuite import PyExperimentSuite
from numpy import *
import os


class MySuite(PyExperimentSuite):

    restore_supported = True

    def reset(self, params, rep):
        # initialize array
        self.numbers = zeros(params['iterations'])

        # seed random number generator
        random.seed(params['seed'])

    def iterate(self, params, rep, n):
        # draw normally distributed random number
        self.numbers[n] = random.normal(params['mean'], params['std'])

        # calculate sample mean and offset
        samplemean = mean(self.numbers[:n+1])
        offset = abs(params['mean']-samplemean)

        # return dictionary
        ret = {'n': n, 'number': self.numbers[n],
            'samplemean': samplemean, 'offset': offset}

        return ret

    def save_state(self, params, rep, n):
        # save array as binary file
```

```
        save(os.path.join(params['path'], params['name'],
            'array_%i.npy'%rep), self.numbers)

    def restore_state(self, params, rep, n):
        # load array from file
        self.numbers = load(os.path.join(params['path'],
            params['name'], 'array_%i.npy'%rep))


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

The configuration file "experiments.cfg" should be in the same folder and contain these lines, defining two experiments:

```
[DEFAULT]
repetitions = 1
iterations = 3000
path = results
seed = None

[normal]
mean = 0
std = 1

[highstd]
mean = 0
std = 5
```

After executing the script with `python suite.py`, the results are stored in the subfolder `/results/normal` and `/results/highstd` respectively. To properly visualize the results, however, we will again use the API to retrieve the results. In this example, we will also make use of the graph and visualization module `matplotlib`, which needs to be installed for the following commands to run. Open a Python shell or create another Python script with these lines:

```
from suite import MySuite
from matplotlib import pyplot as plt
from numpy import *

mysuite = MySuite()
exps = mysuite.get_exps()

print 'lowest offset for experiment normal:', \
    mysuite.get_value(exps[0], 0, 'offset', 'min')
print 'lowest offset for experiment highstd:', \
    mysuite.get_value(exps[1], 0, 'offset', 'min')

# plot results
plt.plot(mysuite.get_history(exps[0], 0, 'offset'), linewidth=2,
    color='blue', label='std=1')
plt.plot(mysuite.get_history(exps[1], 0, 'offset'), linewidth=2,
    color='red', label='std=5')
```

```
plt.xlabel('iterations')
plt.ylabel('offset')
plt.title('sample mean offset to true mean')
plt.legend()
plt.show()
```

The script reads the conducted experiment names from the config file and the requests the histories of the variable `offset` for both experiments, plotting them with the `matplotlib` function `plot()` in different colors.

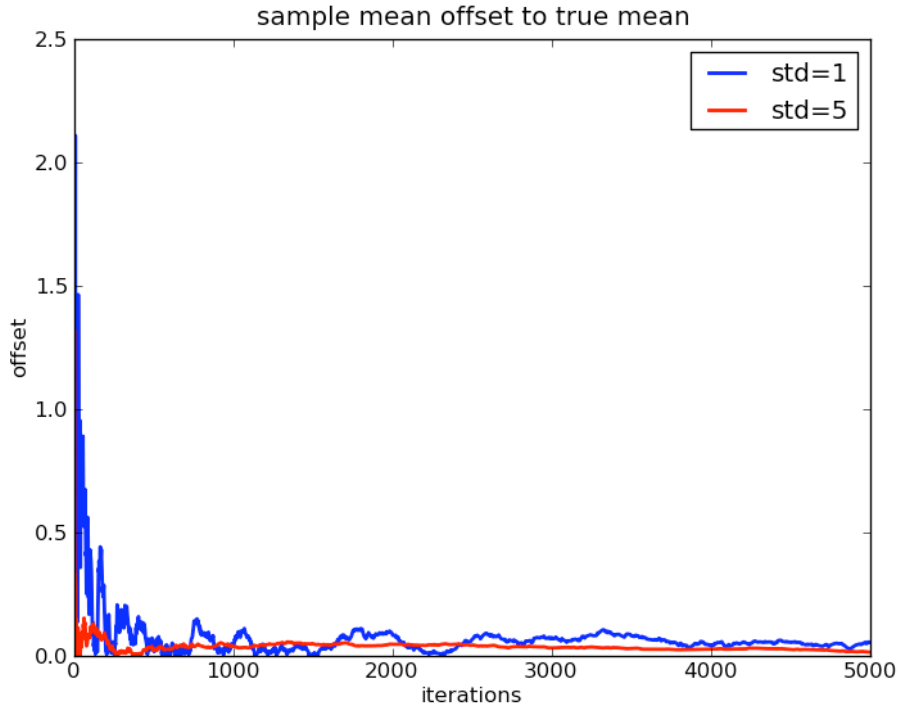

Figure 1. Graph of the experiment histories of the `normal` and `highstd` experiments

Upon running the script, a graph similar to Figure 1 will be displayed. It shows how the offset to the real mean decreases with the number of drawn random numbers for both experiments.

References

Rückstieß, T. and Schmidhuber, J. 2011. A Python Experiment Suite. The Python Papers 6(1): 2.