Artemis Documentation

Release 1.6

Peter O'Connor and Matthias Reisser

Aug 03, 2017

Contents:

CHAPTER 1

Artemis Experiments Documentation

For details on the Experiment API, see *Experiment API*.

A Basic Example

Using this module, you can turn your main function into an "Experiment", which, when run, stores all console output, plots, and computed results to disk (in ~/.artemis/experiments)

For a simple demo, see artemis/fileman/demo_experiments.py

Any function that can be called alone with no arguments can be turned into an experiment using the @experiment_function decorator:

```
from artemis.experiments import experiment_function

@experiment_function
def multiply_3_numbers(a=1, b=2, c=3):
    return a*b*c
```

This turns the function into an Experiment object, which has the following methods:

```
record = multiply_3_numbers.run()
```

Run the function, and save all text outputs and plots to disk. Console output, plots, and result are stored in: ~/.artemis/experiments

```
multiply_3_numbers.add_variant('higher-ab', a=4, b=5)
```

Add a variant to the experiment, with new arguments that override any existing ones. The variant is itself an experiment, and can have variants of its own.

Get a variant of an experiment (or one of its sub-variants).

```
ex = multiply_3_numbers.get_variant('higher-ab')
```

To open up a menu where you can see and run all experiments (and their variants) that have been created run:

```
multiply_3_numbers.browse()
```

This will give you an output that looks something like this:

```
==================== Experiments ====================
  E#  R#    Name                          All Runs                    Duration
↪ Status           Valid    Result
----  ----  ----------------------------  --------------------------  ---------------
↪ ---------------  -------  --------
   0  0     multiply_3_numbers            2017-08-03 10:34:51.150555  0.0213599205017
↪ Ran Succesfully  Yes      6
   1        multiply_3_numbers.higher-ab  <No Records>                -
↪ -                -        -
-----------------------------------------------------
Enter command or experiment # to run (h for help) >>
```

This indicates that you have a saved record of your experiment (created when we called `multiply_3_numbers.run()`), but none of the variant `higher-ab`. In the UI, we can run this variant by entering `run 1`:

```
Enter command or experiment # to run (h for help) >> run 1
```

After running, we will see the status of our experiments updated:

```
==================== Experiments ====================
  E#    R#  Name                          All Runs                      Duration
↪Status           Valid      Result
----  ----  ----------------------------  --------------------------  ----------  ----
↪-----------  -------  --------
   0     0  multiply_3_numbers            2017-08-03 10:34:51.150555   0.0213599  Ran
↪Succesfully  Yes             6
   1     0  multiply_3_numbers.higher-ab  2017-08-03 10:38:45.836260   0.0350862  Ran
↪Succesfully  Yes             60
-----------------------------------------------------
Enter command or experiment # to run (h for help) >>
```

From the UI we have access to a variety of commands for showing and comparing experiments. For example, *argtable* prints a table comparing the results of the different experiments:

```
Enter command or experiment # to run (h for help) >> argtable all
    ------------------------------------------------------  ------------------  -----
↪----------  -----------  --------------  ------
                                                            Function            Run
↪Time         Common Args  Different Args  Result
    2017.08.03T10.34.51.150555-multiply_3_numbers           multiply_3_numbers  0.
↪0213599205017  c=3          a=1, b=2        6
    2017.08.03T10.38.45.836260-multiply_3_numbers.higher-ab  multiply_3_numbers  0.
↪0350861549377  c=3          a=4, b=5        60
    ------------------------------------------------------  ------------------  -----
↪----------  -----------  --------------  ------
```

More Examples

- An example demonstrating Artemis's Experiment framework on a simple MNIST classification task
- Step-by-step tutorial on using Artemis to organize your Experiments

CHAPTER 2

Experiment API

You can use the UI for most things related to running and viewing the results of experiments. (See *Artemis Experiments Documentation* for how to do that).

This document shows the methods for interacting programatically with the experiment interface.

Creating Experiments

Experiment Decorators are turn your python functions into experiments. When using decorators, **be sure that your experiment function is uniquely named** (ie, no other function in your code has the same name). This is important because when results are saved, the name of the function is used to identify what experiment the results belong to.

`artemis.experiments.`**`experiment_function`**(*f*)
Use this decorator (@experiment_function) on a function that you want to run. e.g.

```
@experiment_function
def demo_my_experiment(a=1, b=2, c=3):
    ...
```

This turns your function demo_my_experiment into an experiment. It can still be called as a normal function, but it now has can also be called with the methods of an Experiment object (eg. demo_my_experiment.run()).

`artemis.experiments.`**`experiment_root`**(*f*)
Use this decorator on a function that you want to build variants off of:

```
@experiment_root
def demo_my_experiment(a, b=2, c=3):
    ...
```

The root experiment is not runnable by itself, and will not appear in the list in the browse experiments UI, but you can call `demo_my_experiment.add_variant(...)` to create runnable variants.

class artemis.experiments.**ExperimentFunction**(*display_function=None*, *comparison_function=None*, *one_liner_function=None*, *is_root=False*)

This is the most general decorator. You can use this to add details on the experiment.

__init__(*display_function=None*, *comparison_function=None*, *one_liner_function=None*, *is_root=False*)

Parameters

- **display_function** – A function that takes the results (whatever your experiment returns) and displays them.
- **comparison_function** – A function that takes an OrderedDict<experiment_name, experiment_return_value>. You can optionally define this function to compare the results of different experiments. You can use call this via the UI with the compare_experiment_results command.
- **one_liner_function** – A function that takes your results and returns a 1 line string summarizing them.
- **is_root** – True to make this a root experiment - so that it is not listed to be run itself.

artemis.experiments.**capture_created_experiments**(**args*, ***kwds*)

A convenient way to cross-breed experiments. If you define experiments in this block, you can capture them for later use (for instance by modifying them). e.g.:

```
@experiment_function
def add_two_numbers(a=1, b=2):
    return a+b

with capture_created_experiments() as exps:
    add_two_numbers.add_variant(a=2)
    add_two_numbers.add_variant(a=3)

for ex in exps:
    ex.add_variant(b=4)
```

Return type Generator[Experiment]

The Experiment

The above decorators return Experiment Objects, which have the following API...

class artemis.experiments.experiments.**Experiment**(*function=None*, *display_function=None*, *comparison_function=None*, *one_liner_function=None*, *name=None*, *is_root=False*)

An experiment. In general you should not use this class directly. Use the experiment_function decorator, and create variants using decorated_function.add_variant()

add_root_variant(*variant_name=None*, ***kwargs*)

Add a variant to this experiment, but do NOT register it on the list of experiments. There are two ways you can do this:

```
# Name the experiment explicitely, then list the named arguments
my_experiment_function.add_root_variant('big_a', a=10000)
assert my_experiment_function.get_name()=='my_experiment_function.big_a'
```

```
# Allow the experiment to be named automatically, and just list the named␣
↪arguments
my_experiment_function.add_root_variant(a=10000)
assert my_experiment_function.get_name()=='my_experiment_function.a=10000'
```

Parameters

- **variant_name** – Optionally, the name of the experiment
- **kwargs** – The named arguments which will differ from the base experiment.

Returns The experiment.

add_variant(*variant_name=None*, ***kwargs*)

Add a variant to this experiment, and register it on the list of experiments. There are two ways you can do this:

```
# Name the experiment explicitely, then list the named arguments
my_experiment_function.add_variant('big_a', a=10000)
assert my_experiment_function.get_name()=='my_experiment_function.big_a'

# Allow the experiment to be named automatically, and just list the named␣
↪arguments
my_experiment_function.add_variant(a=10000)
assert my_experiment_function.get_name()=='my_experiment_function.a=10000'
```

Parameters

- **variant_name** – Optionally, the name of the experiment
- **kwargs** – The named arguments which will differ from the base experiment.

Returns The experiment.

get_all_variants(*include_roots=False*, *include_self=True*)

Return a list of variants of this experiment :param include_roots: Include "root" experiments :param include_self: Include this experiment (unless include_roots is false and this this experiment is a root) :return: A list of experiments.

get_args()

Parameters **to_root** – If True, find all args of this experiment down to the root experiment. If False, just return the args that differentiate this variant from its parent.

Returns A dictionary of arguments to the experiment

get_latest_record(*only_completed=False*, *err_if_none=True*)

Return the ExperimentRecord from the latest run of this Experiment.

Parameters

- **only_completed** – Only search among records of that have run to completion.
- **err_if_none** – If True, raise an error if no record exists. Otherwise, just return None in this case.

Returns An ExperimentRecord object

get_variant(*variant_name=None*, ***kwargs*)

Get a variant on this experiment.

Parameters

- **variant_name** – The name of the variant, if it has one
- **kwargs** – Otherwise, the named arguments which were used to define the variant.

Returns An Experiment object

get_variant_records(*only_completed=False*, *only_last=False*, *flat=False*)
Get the collection of records associated with all variants of this Experiment.

Parameters

- **only_completed** – Only search among records of that have run to completion.
- **only_last** – Just return the most recent record.
- **flat** – Just return a list of records

Returns if not flat (default) An OrderedDict<experiment_id: ExperimentRecord>. otherwise, if flat: a list<ExperimentRecord>

has_record(*completed=True*, *valid=True*)
Return true if the experiment has a record, otherwise false. :param completed: Check that the record is completed. :param valid: Check that the record is valid (arguments match current experiment arguments) :return: True/False

run(*print_to_console=True*, *show_figs=None*, *test_mode=None*, *keep_record=None*, *raise_exceptions=True*, *display_results=True*, ***experiment_record_kwargs*)
Run the experiment, and return the ExperimentRecord that is generated.

Parameters

- **print_to_console** – Print to console (as well as logging to file)
- **show_figs** – Show figures (as well as saving to file)
- **test_mode** – Run in "test_mode". This sets the global "test_mode" flag when running the experiment. This flag can be used to, for example, shorten a training session to verify that the code runs. Can be: True: Run in test mode False: Don't run in test mode: None: Keep the current state of the global "is_test_mode()" flag.
- **keep_record** – Keep the folder that results are saved into. True: Results are saved into a folder False: Results folder is deleted at the end. None: If "test_mode" is true, then delete results at end, otherwise save them.
- **raise_exceptions** – True to raise any exception that occurs when running the experiment. False to catch it, print the error, and move on.
- **experiment_record_kwargs** – Passed to the "record_experiment" context.

Returns The ExperimentRecord object, if keep_record is true, otherwise None

The Experiment Record

When you run an Experiment, a folder is created in which the stdout, results, figures, and other info are stored. The ExperimentRecord object provides an API for accessing the contents of this folder.

class `artemis.experiments.experiment_record.`**ExperimentRecord**(*experiment_directory*)
A Record of a run of an Experiment. This object allows you to access data stored in the directory that was created for that run of the experiment. Experiment Records are stored in `~/artemis/experiments/`.

delete()
Delete this experiment record from disk.

get_args()
Get the arguments with which this record was run. :return: A list of 2-tuples of (arg_name, arg_value)

get_error_trace()
Get the error trace, or return None if there is no error trace. :return:

get_experiment()
Load the experiment associated with this record. Note that this will raise an ExperimentNotFoundError if the experiment has not been imported. :return: An Experiment object

get_id()
Get the id of this experiment record. Generally in format '<datetime>-<experiment_name>' :return:

get_log()

Returns The stdout generated during the run of this experiment.

info

Returns An ExperimentRecordInfo object, containing info about the experiment (name, runtime, etc)

is_valid(*last_run_args=None*, *current_args=None*)

Returns True if the experiment arguments have not changed False if they have changed None if it cannot be determined because arguments are not hashable objects.

list_files(*full_path=False*)
List files in experiment directory, relative to root. :param full_path: If true, list file with the full local path :return: A list of strings indicating the file paths.

load_figures()

Returns A list of matplotlib figures generated in the experiment. The figures will not be drawn yet, so you will have to call plt.show() to draw them or plt.draw() to draw them.

open_file(*filename*, **args*, ***kwargs*)
Open a file within the experiment record folder. Example Usage:

with record.open_file('filename.txt') as f: txt = f.read()

Parameters

- **filename** – Path within experiment directory (it can include subdirectories)
- **kwargs** (*args,*) – Forwarded to python's "open" function

Returns A file object

show_figures(*hang=False*)
Show all figures that were saved during the run of the experiment. :param hang: If True, and figures were saved matplotlib figures, hang execution until they are closed.

CHAPTER 3

Artemis Plotting

Live Plots with dbplot

`dbplot` is an easy way to create a live plot of your data.

For example, to create live updating plots of a random grid:

```
from artemis.plotting.db_plotting import dbplot
import numpy as np

for _ in xrange(50):
    dbplot(np.random.randn(20, 10), 'random data')
```

A plot will come up showing the random data.

```
from artemis.plotting.db_plotting import dbplot
import numpy as np

for _ in xrange(50):
    dbplot(np.random.randn(20, 10), 'random line data', plot_type='line')
```

You can include multiple plots:

```
from artemis.plotting.db_plotting import dbplot
import numpy as np

for _ in xrange(50):
    dbplot(np.random.randn(20, 2), 'random line data', plot_type='line')
    dbplot(np.random.randn(10, 10), 'random grid data')
```

If you plot many things, you may want to "hold" your plots, so that they all update together. This speeds up the rate of plotting:

```
from artemis.plotting.db_plotting import dbplot, hold_dbplots
import numpy as np
```

```
for _ in xrange(50):
    with hold_dbplots():
        dbplot(np.random.randn(20, 2), 'random line data', plot_type='line')
        dbplot(np.random.randn(10, 10), 'random grid data')
        dbplot(np.random.randn(4), 'random line history')
        dbplot(np.random.rand(3), 'random bars', plot_type='bar')
        dbplot([np.random.rand(20, 20), np.random.randn(20, 16, 3)], 'multi image')
```

dbplot documentation

artemis.plotting.db_plotting.**dbplot**(*data*, *name=None*, *plot_type=None*, *axis=None*, *plot_mode='live'*, *draw_now=True*, *hang=False*, *title=None*, *fig=None*, *xlabel=None*, *ylabel=None*, *draw_every=None*, *layout=None*, *legend=None*, *grid=False*, *wait_for_display_sec=0*, *cornertext=None*)

Plot arbitrary data and continue execution. This program tries to figure out what type of plot to use.

Parameters

- **data** – Any data. Hopefully, we at dbplot will be able to figure out a plot for it.
- **name** – A name uniquely identifying this plot.
- **plot_type** – A specialized constructor to be used the first time when plotting. You can also pass certain string to give hints as to what kind of plot you want (can resolve cases where the given data could be plotted in multiple ways): 'line': Plots a line plot 'img': An image plot 'colour': A colour image plot 'pic': A picture (no scale bars, axis labels, etc).
- **axis** – A string identifying which axis to plot on. By default, it is the same as "name". Only use this argument if you indend to make multiple dbplots share the same axis.
- **plot_mode** – Influences how the data should be used to choose the plot type: 'live': Best for 'live' plots that you intend to update as new data arrives 'static': Best for 'static' plots, that you do not intend to update 'image': Try to represent the plot as an image
- **draw_now** – Draw the plot now (you may choose false if you're going to add another plot immediately after and don't want have to draw this one again.
- **hang** – Hang on the plot (wait for it to be closed before continuing)
- **title** – Title of the plot (will default to name if not included)
- **fig** – Name of the figure - use this when you want to create multiple figures.
- **grid** – Turn the grid on
- **wait_for_display_sec** – In server mode, you can choose to wait maximally wait_for_display_sec seconds before this call returns. In case plotting is finished earlier, the call returns earlier. Setting wait_for_display_sec to a negative number will cause the call to block until the plot has been displayed.

Plotting Demos

- A demo of showing how to make various kinds of live updating plots.
- A demo repo showing how to use Artemis from your code

- A guide on using Artemis for remote plotting

Browser-Plotting

After installing, you should have a file `~/.artemisrc`.

To use the web backend, edit the `backend` field to `matplotlib-web`.

To try it you can run the commands described above for dbplot.

Artemis is a collection of tools that make it easier to run experiments in Python. These include:

- An easy-to-use system for making live plots, to monitor variables in a running experiment.
- A browser-based plotter for displaying live plots.
- A framework for defining experiments and logging their results (text output and figures) so that they can be reviewed later and replicated easily.
- A system for downloading/caching files, to a local directory, so the same code can work on different machines.

To install Artemis, see Artemis on Github

Index

Symbols

A

C

D

E

G

H

I

L

O

R

S