experipy Documentation

Release 0.2.0

Adam Howard

Sep 14, 2020

Contents:

experipy is a framework for writing and running Computational Science experiments. It provides facilities for describing an experiment as a shell script, and mechanisms for then running it. Experiments can be run locally and also submitted to a cluster's job queuing system as a PBS script.

```
from experipy.exp       import Experiment
from experipy.grammar   import Executable

echo = Executable("echo",
    ["Hello World", "> test.out"],
    outputs=["test.out"]
)

exp = Experiment(echo, expname="test", destdir="results")
exp.run()
```

The intention of experipy is to act as the core of a researcher's scripting framework. In the author's research group, projects often involved running dozens of benchmarks with hundreds of configurations in parallel across a cluster, so experipy was designed to ease the design and scripting of new experiments and configurations.

CHAPTER 1

Installation

`experipy` is available on PyPI:

```
pip install experipy
```

Or, you can find it on Github at https://github.com/Elemnir/experipy.

1.1 1. experipy.grammar - Composing experiments

This module provides the core elements which compose the Experipy grammar: Executables, Wrappers, Pipelines, and Groups. These elements facilitate specifying programs to execute as well as the files they depend on.

1.1.1 1.1. Element objects

class `experipy.grammar.`**`Element`**(*inputs=None*, *outputs=None*)

The Element class forms the grammar's base class.

Parameters

- **`inputs`** (*`list`*) – A list of strings which are the names of files that the Element relies on for input. These will be copied to the run directory when an Experiment is used to run the Element.
- **`outputs`** (*`list`*) – A list of strings which are the names of files that the Element is expected to generate as output. These will be copied from the run directory when an Experiment is used to run the Element.

`inputs`()

Generator which yields the Element's input files

`outputs`()

Generator which yields the Element's output files

1.1.2 1.2. Executable objects

The Executable class extends the base Element class by providing an abstraction for describing a program executable. Once instantiated, converting an Executable object to a string will yield the command string that will be entered into the shell script.

class experipy.grammar.**Executable**(*prog*, *opts=None*, *wait=True*, ***kwargs*)
Executable objects should represent a single program and its arguments.

Parameters

- **prog** (*str*) – The name of the program executable.
- **opts** (*list*) – A list of command line options to pass to the program. Defaults to an empty list if not provided.
- **wait** (*bool*) – If False, a '&' will be appended to the argument list, indicating to the shell that it should background the program instead of blocking on it. Defaults to True.

1.1.3 1.3. Wrapper objects

Wrappers are executables which accept another Executable and its arguments as a parameter, and incorporates the wrapped Executable into its argument list and collection of inputs and outputs.

class experipy.grammar.**Wrapper**(*prog*, *opts*, *wrapped*, ***kwargs*)
Wrapper objects allow specification of a program which wraps another.

Wrappers are a subclass of Executable which allow specification of programs such as GDB or Valgrind, which wrap around another program to alter or observe its execution.

Parameters

- **prog** (*str*) – The name of the program executable.
- **opts** (*list*) – A list of command line options to pass to the program. Must minimally contain a string having the value '[[wrapped]]', which indicates where the wrapped executable should be inserted into the wrapping executable's argument list.
- **wrapped** (*experipy.Executable*) – The wrapped Executable. Inputs and outputs specified to wrapped will be included in the resultant object's inputs and outputs.
- **wait** (*bool*) – If False, a '&' will be appended to the argument list, indicating to the shell that it should background the program instead of blocking on it. Defaults to True.

inputs()
Generator which yields the Wrapper's input files

outputs()
Generator which yields the Wrapper's output files

1.1.4 1.4. Pipeline objects

The Linux shell supports piping of output from one program into the input of another. Pipelines provide a mechanism to support that feature in the generated shell scripts.

class experipy.grammar.**Pipeline**(**parts*, ***kwargs*)
Pipeline objects allow specification of pipelined workflows.

A Pipeline takes one or more Element parts, and joins them with a 'l' operator, indicating to the shell that each part should recieve its input from the previous part, and provide its output to the next.

Parameters ***parts** – One or more Executables or Wrappers to be chained together into a pipeline. Inputs and outputs to the individual parts will be included in the Pipeline's inputs and outputs.

inputs()
Generator which yields the Pipeline's input files

outputs()
Generator which yields the Pipeline's output files

1.1.5 1.5. Group objects

Groups allow generation of more complex experiment behavior than the execution of a single Executable, Wrapper, or Pipeline.

class experipy.grammar.**Group**(**parts*, ***kwargs*)
Group objects allow specification of Executables to be run in order.

In the resultant script, a Group's parts will be included one after another, in the order they were specified. Groups should be used when specifying complex experiments involving multiple steps like set up or post-processing, or combined with the wait parameter to Executable to specify programs which should be run concurrently. A Group can also be used as a part in another Group.

Parameters ***parts** – One or more Elements to be placed into the script. Inputs and outputs to the individual parts will be included in the Group's inputs and outputs.

inputs()
Generator which yields the Group's input files

outputs()
Generator which yields the Group's output files

1.1.6 1.6. Block objects

Blocks are simple text blocks that will be rendered into the runscript without additional processing.

class experipy.grammar.**Block**(*text*, ***kwargs*)
Blocks allow arbitrary text to be rendered into the script without further processing or enforcing other grammatical rules.

Parameters **text** (*str*) – The text to be rendered.

1.2 2. experipy.exp - The Experiment Runner

This module provides the Experiment class for running compositions in the grammar, as well as the Exp Namespace for controlling and configuring Experiment behavior.

1.2.1 2.1. An Example

```
from experipy.exp       import Experiment
from experipy.grammar   import Executable

exp = Experiment(Executable("echo", ["Hello World"]),
                 expname="test",
```

```
                destdir="results")
exp.run()
```

This will run the program `echo` with the argument `Hello World` in a directory in `/tmp`, writing the output and error, along with timing information, to the directory `results`. Directories will be created as needed.

1.2.2 2.2. Experiment objects

class `experipy.exp.`**`Experiment`**(*cmd*, *expname='exp'*, *destdir=None*)

Experiment objects perform the generation and execution of runscripts.

Once a composition has been specified in the grammar, wrapping it in an Experiment allows the user to generate a shell script as a string using the make_runscript method. The run and queue methods provide mechanisms for executing the generated scripts.

Parameters

- **`cmd`** (*`experipy.Element`*) – A composition of experipy Elements such as Executable and Group, which defines the behavior the user wishes the Experiment to perform.
- **`expname`** (*`str`*) – A name to be used for identifying the experiment. Defaults to Exp.defname, which defaults to "exp".
- **`destdir`** (*`str`*) – An optional path to a directory where the results from running the experiment should be stored. If None, expname will be used.

`make_runscript`(*preamble='#!/bin/bash'*, *rm_rundir=True*)

Create a string containing the experiment rendered as a shell script.

Parameters

- **`preamble`** (*`str`*) – The first line(s) of the runscript. Defaults to Exp.shebang, which defaults to "#!/bin/bash".
- **`rm_rundir`** (*`bool`*) – If True, a line deleting the experiment's working directory will be added to the end of the script. Defaults to True.

Returns A run script as described by the composition provided to the Experiment.

Return type str

`queue`(*h=False*, *n=False*, *q=None*, *A=None*, ***kwargs*)

Submit the experiment to a job queuing system as a PBS script.

Generates a script with a PBS script header, writes the script to the results directory, and then submits it to the job queuing system by running the command qsub as a subprocess.

Parameters

- **`h`** (*`bool`*) – Will add a `-h` to pbs headers if True, Default is False.
- **`n`** (*`bool`*) – Will add a `-n` to pbs headers if True, Default is False.
- **`q`** (*`str`*) – Optionally request a resource queue.
- **`A`** (*`str`*) – Optionally name the account to charge for this job.
- **`**kwargs`** – The remaining keyword arguments will be combined into resource requests with -l.

`run`(*rm_rundir=True*)

Execute the experiment as a subprocess of the current process.

Generates a run script, writes that script to the results directory, and then executes the script as a subprocess of the current process. The time the script takes to execute, including setup and clean up time, is recorded. This function blocks until the experiment is complete.

Parameters **rm_rundir** (*bool*) – If True, the directory created for running the experiment will be deleted at the end of the experiment. Defaults to True.

sbatch(***kwargs*)

Submit the experiment to a Slurm cluster as an sbatch script.

Generates a script with a Slurm script header, writes the script to the results directory, and then submits it to the job queuing system by running the command sbatch as a subprocess.

Parameters ****kwargs** – Keyword arguments will be translated to SBATCH directives of the form `#SBATCH --<key>=<value>`. Underscores in keyword argument names will be substituted for dashes in the emitted SBATCH directives. For example, `cpus_per_task=4` will be translated to `#SBATCH --cpus-per-task=4`.

1.2.3 2.3. The Exp Namespace

Default values for paths and filenames in the Experiment class are controlled by a Namespace called `Exp`. These defaults are listed below, and can be overridden by setting a new value in the `.experipyrc` under the `[Exp]` section.

Key	Default Value	Description
shebang	#!/bin/bash	The first line of the generated shell scripts.
rundir	/tmp	Path to the directory where the experiment is going to be run.
defname	exp	Default name of experiments.
runsh	run.sh	Name of the generated shell scripts.
out	raw.out	Name of the file which will collect the experiment's standard output.
err	raw.err	Name of the file which will collect the experiment's standard error.
timing	harness_time.out	When an experiment is run using `run()`, its run time will be captured in this file.

1.3 3. experipy.system - System tools in the grammar

This module provides a number of system and shell tools for helping to specify common tasks within the experipy grammar.

`experipy.system.`**cd**(*dirname*)

`experipy.system.`**cp**(*target*, *dest*, *opts=[]*)

`experipy.system.`**mkdir**(*dirname*, *make_parents=False*)

`experipy.system.`**mkfifo**(*pipename*)

`experipy.system.`**rm**(**files*)

`experipy.system.`**wait**()

`experipy.system.`**python_script**(*script*, *sopts=[]*, *pythonexe='python'*, ***kwargs*)

`experipy.system.`**java_app**(*jarfile*, *popts=[]*, *javaexe='java'*, *jopts=[]*, ***kwargs*)

1.4 4. experipy.config - Configuration utilities

This modules provides the Namespace class, which provides a mechanism for defining collections of configurable constants.

1.4.1 4.1. Namespace objects

class `experipy.config.`**`Namespace`**(*name=None*, ***kwargs*)

Namespace objects are intended to act as collections of constants.

All arguments passed to the Namespace when it is instantiated are bound to attributes of the instance, allowing attribute reference as opposed to dictionary access syntax. For example: `n = Namespace("N", foo="bar")` would generate a namespace with an attribute `n.foo` whose value is `"bar"`.

Namespaces also support configuration using configparser INI files. By default, configuration is stored and read from `~/.experipyrc`, unless the environment variable `EXPERIPY_CONFIG_PATH` is set, in which case that value is used as the filename.

Parameters

- **`name`** (*str*) – The name to assign to the namespace. If not provided, the resulting Namespace instance will be anonymous and not configurable via the configuration file.
- **`**kwargs`** – The remaining keyword arguments will be added the Namespace's dictionary, allowing for attribute access. If a name was provided, and the namespace had a section in the configuration file, conflicting arguments will have their values ignored in favor of the value in the configuration file.

classmethod **`dump_full_config`**(*fname='/home/docs/.experipyrc'*)

Write a config of all instantiated and preconfigured Namespaces.

All instantiated and named Namespaces will be dumped to the given file, along with any Namespace configurations which have been loaded from the config, but whose corresponding Namespace has not yet been instantiated.

Parameters **`fname`** (*str*) – Name of the file to write the config to. If not provided, it will default to the current config file ("~/.experipyrc" or the value of the EXPERIPY_CONFIG_PATH environment variable).

1.5 5. experipy.metrics - Results Parsing

This module provides the Metric class as a means of defining and extracting values from the results of Experiment runs.

1.5.1 5.1. Metric objects

class `experipy.metrics.`**`Metric`**(*name*, *filename*, *regex*, *parser=<type 'float'>*)

Metric objects define a value to be extracted from a given file.

A metric consists of a base filename, a regex with which to search that file, and a parser which converts the value, once found, into the desired type.

Parameters

- **`name`** (*str*) – The name of the metric.

- **filename** (*str*) – The name of the file in a given results directory to search. For instance, if the metric should appear in the standard output of a given experiment, then filename should be set as `raw.out`.
- **regex** (*str*) – A string which will be compiled as a regular expression and used to search for the metric. Must contain a Named Group with the name `value` (i.e. `(?P<value>\d+)`).
- **parser** (*callable*) – A callable taking a single string argument and returning the value converted to the desired type. Defaults to `float`.

get_value(*resultpath*, *default=None*)

Given a path to a results directory, attempt to extract the value. Optionally provide a default value in the event the value can't be found.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index