compapp Documentation

Release 0

Takafumi Arakaki

Apr 08, 2017

Contents

Contents:

CHAPTER 1

What is compapp?

Automatic data directory naming, creation and management

When writing programs for numerical simulations and data analysis, managing directories to store resulting data (called *datastore* in this document) is hard.

`Computer` & `Executable` subclasses	`datastore` property	Behavior
`Computer`	`DirectoryDataStore`	Simulation is run with a specified data directory in which simulation parameters and results are saved. *Nested classes* such as `Plotter` and other nested `Computer`s may use sub-paths.
nested `Computer`	`DirectoryDataStore`	If a `Computer` subclass nests in some *owner app*, `DirectoryDataStore` automatically allocates sub-directory under the directory used by the *owner app*.
`Plotter`, `Loader`	`SubDataStore`	Use files under the directory of the *owner app*.
`Memoizer`	`HashDataStore`	Data analysis is run with a data directory automatically allocated depending on the parameter values (including data files). The rationale here is that data analysis has to yield the same result given parameters. Thus, if the datastore already exists when this application is run, it loads the results rather than re-computing them. In other words, combinations of `Memoizer` act as build dependencies defined by Makefile and similar build tools. Since generated datastore path is not human friendly (it is based on hash), compapp provides command line interface to help house-keeping.

Parameter management

Simulations and data analysis require various parameters for each run. Those parameters often have nested sub-parameters reflecting sub-simulations and sub-analysis. compapp naturally supports such nested parameters using *nested class*. See `Parametric`.

When parameters have deeply nested structure, it is hard to run a simulation or analysis with slightly different parameters. `Computer.cli` provides a CLI to set such "deep parameters" on the fly.

Automatic type-check and value-check for properties (traits)

Simulations and data analysis require certain type of parameters but checking them manually is hard and letting an error to happen at the very end of long-running computations is not an option. compapp provides a very easy way to configure such type checks. The main idea implemented in `Parametric` is that, for simple Python data types, the default values define required data type:

```
>>> from compapp import Parametric
>>> class MyParametric(Parametric):
...     i = 1
...     x = 2.0
>>> MyParametric(i=1.0)
Traceback (most recent call last):
    ...
ValueError: Value 1.0 (type: float) cannot be assigned to the variable
MyParametric.i (default: 1) which only accepts one of the following types:
int, int16, ...
>>> MyParametric(x='2.0')
Traceback (most recent call last):
    ...
ValueError: Value '2.0' (type: str) cannot be assigned to the variable
MyParametric.x (default: 2.0) which only accepts one of the following types:
float, float16, ...
```

For more complex control, there are descriptors such as `Instance`, `Required`, `Optional`, etc. Collection-type descriptors such as `List` and `Dict` restricts data types of its component (e.g., dict key has to be a string and the value has to be int) and other traits such as maximal length. The descriptor `Choice` restricts the *value* of properties, rather than the type. The descriptor `Or` defines a property that must satisfy one of defined restrictions.

Linking properties

compapp prefers *composition over inheritance*. However, using composition makes it hard to share properties between objects whereas in inheritance it is easy (or too easy[1]) to share properties between parent and sub classes. compapp provides various *linking properties* (`Link`, `Delegate`, etc.) which can refer to properties of other objects.

Hooks

`Executable` defines various methods *to be extended* where user's simulation and data analysis classes can hook some computations. User should at least extend the `run` method to implement some computations. Although methods `save` and `load` can also be extended, `AutoDump` plugin can handle saving and loading results and parameters automatically. There are `prepare` and `finish` methods to be called always, not depending on whether the executable class is `run` or `load`ed.

See also: API Reference

[1] In other words, sharing properties is opt-in for composition approach and forced for inheritance approach.

Plugins

`Executable` (hence `Computer`) provides various hooks so that it is easy to "inject" some useful functions via plugins. In fact, the main aim of compapp is to provide well-defined set of hooks and a system for easily coordinating different components by *linking properties*.

Here is the list of plugins provided by `compapp.plugins`:

`recorders.DumpResults`	Automatically save owner's results.
`recorders.DumpParameters`	Dump parameters used for its owner.
`timing.RecordTiming`	Record timing information.
`vcs.RecordVCS`	Record VCS revision automatically.
`misc.Logger`	Interface to pre-configured `logging.Logger`.
`misc.Debug`	Debug helper plugin.
`misc.Figure`	A wrapper around `matplotlib.pyplot.figure`.
`datastores.DirectoryDataStore`	Data-store using a directory.
`datastores.SubDataStore`	Data-store using sub-paths of parent data-store.
`datastores.HashDataStore`	Automatically allocated data-store based on hash of parameter.

CHAPTER 2

Tutorial

```
>>> import os
>>> import numpy
>>> from compapp import Computer
```

```
>>> class Sine(Computer):
...     steps = 100
...     freq = 50.0
...     phase = 0.0
...
...     def run(self):
...         ts = numpy.arange(self.steps) / self.freq + self.phase
...         self.results.xs = numpy.sin(2 * numpy.pi * ts)
```

Call `execute` (*not* `run`) to run the computation:

```
>>> app = Sine()
>>> app.execute()
>>> app.results.xs
array([...])
```

Any attributes assigned to `results` are going to be saved in `datastore.dir` if it is specified:

```
>>> app = Sine()
>>> app.datastore.dir = 'out'
>>> app.execute()
>>> npz = numpy.load('out/results.npz')
>>> numpy.testing.assert_equal(app.results.xs, npz['xs'])
```

You can also pass (nested) dictionary to the class:

```
>>> app = Sine({'datastore': {'dir': 'another-dir'}})
>>> app.datastore.dir
'another-dir'
```

Plotting

The `figure` attribute of `Computer` is a simple wrapper of `matplotlib.pyplot`.

```
>>> class MyApp(Computer):
...     sine = Sine
...
...     def run(self):
...         self.sine.execute()
...         _, ax = self.figure.subplots()  # calls pyplot.subplots()
...         ax.plot(self.sine.results.xs)
```

```
>>> app = MyApp()
>>> app.datastore.dir = 'out'
>>> app.execute()
```

The plot is automatically saved to a file in the *datastore* directory:

```
>>> os.path.isfile('out/figure-0.png')
True
```

In interactive environments, the figures are also shown via default matplotlib backend (e.g., as inline figures in Jupyter/IPython notebook), provided that `setup_interactive` is called first.

Composition of computations

Since `MyApp` is built on top of `Sine`, the result of `Sine` is also saved in the datastore of `MyApp`.

```
>>> os.path.isfile('out/sine/results.npz')
True
```

The parameter passed to the root class is passed to nested class:

```
>>> app = MyApp({'sine': {'phase': 0.5}})
>>> app.sine.phase
0.5
```

Decomposing parameters and computations in reusable building blocks makes code simple.

For example, suppose you want to try many combinations of frequencies and phases. You can use `numpy.linspace` for this purpose. Naive implementation would be like this:

```
class NaiveMultiSine(Computer):
    steps = 100

    freq_start = 10.0
    freq_stop = 100.0
    freq_num = 50

    phase_start = 0.0
    phase_stop = 1.0
    phase_num = 50

    def run(self):
        freqs = numpy.linspace(
```

```
            self.freq_start, self.freq_stop, self.freq_num)
        phases = numpy.linspace(
            self.phase_start, self.phase_stop, self.phase_num)
        ...
```

A better way is to use `Parametric` and make a composable part:

```
>>> from compapp import Parametric
>>> class LinearSpace(Parametric):
...     start = 0.0
...     stop = 1.0
...     num = 50
...
...     @property
...     def array(self):
...         return numpy.linspace(self.start, self.stop, self.num)
```

Then `LinearSpace` can be used as attributes:

```
>>> class MultiSine(Computer):
...     steps = 100
...     phases = LinearSpace
...
...     class freqs(LinearSpace):  # subclass to change default start/stop
...         start = 10.0
...         stop = 100.0
...
...     def run(self):
...         freqs = self.freqs.array
...         phases = self.phases.array
...
...         ts = numpy.arange(self.steps)
...         xs = numpy.zeros((len(freqs), len(phases), self.steps))
...         for i, f in enumerate(freqs):
...             for j, p in enumerate(phases):
...                 xs[i, j] = numpy.sin(2 * numpy.pi * (ts / f + p))
...         self.results.xs = xs
...
>>> app = MultiSine()
>>> app.freqs.num = 10
>>> app.phases.num = 20
>>> app.execute()
>>> app.results.xs.shape
(10, 20, 100)
```

Dynamic loading

You can switch a part of computation at execution time:

```
>>> class Cosine(Sine):
...     def run(self):
...         ts = numpy.arange(self.steps) / self.freq + self.phase
...         self.results.xs = numpy.cos(2 * numpy.pi * ts)
```

```
>>> from compapp import dynamic_class
>>> class MyApp2(Computer):
...     signal, signal_class = dynamic_class('.Sine', __name__)
...
...     def run(self):
...         self.signal.execute()
...         _, ax = self.figure.subplots()
...         ax.plot(self.signal.results.xs)
...
>>> assert isinstance(MyApp2().signal, Sine)
>>> assert isinstance(MyApp2({'signal_class': '.Cosine'}).signal, Cosine)
```

Trying out multiple parameters (in parallel)

To vary parameters of a computation, you can use the CLI bundled with compapp:

```
capp mrun DOTTED.PATH.TO.A.CLASS -- \
    '--builder.ranges["PATH.TO.A.PARAM"]:leval=(START,[ STOP[, STEP]])' \
    '--builder.linspaces["PATH.TO.A.PARAM"]:leval=(START,[ STOP[, STEP]])' \
    '--builder.logspaces["PATH.TO.A.PARAM"]:leval=(START,[ STOP[, STEP]])' \
    ...
```

You can also use the same functionality in Python code:

```
>>> from compapp import Variator
>>> class MyVariator(Variator):
...     base, classpath = dynamic_class('.MyApp', __name__)
...
...     class builder:
...         linspaces = {
...             'sine.freq': (10.0, 100.0, 50),
...             'sine.phase': (0.0, 1.0, 50),
...         }
...
>>> app = MyVariator()
>>> app.builder.linspaces['sine.freq'] = (10.0, 100.0, 3)  # num = 3
>>> app.builder.linspaces['sine.phase'] = (0.0, 1.0, 2)  # num = 2
>>> app.execute()
>>> len(app.variants)  # = 3 * 2
6
>>> assert isinstance(app.variants[0], MyApp)
```

CHAPTER 3

Examples

Old examples

Simple example

```
from compapp import Computer


class SimpleApp(Computer):

    x = 1.0
    y = 2.0

    def run(self):
        self.results.sum = self.x + self.y


if __name__ == '__main__':
    app = SimpleApp.cli()
```

Run:

```
python simple.py --datastore.dir OUT
```

Result:

```
OUT/results.json
```

Simple plotting example

```
from compapp import Computer


class PlottingApp(Computer):

    def run(self):
        fig = self.figure()
        ax = fig.add_axes(111)
        ax.plot([1, 2, -1, 5])


if __name__ == '__main__':
    app = PlottingApp.cli()
```

Run:

```
python simple_plotting.py --datastore.dir OUT
```

Result:

```
OUT/figure-0.png
```

CHAPTER 4

Executables

Kinds of computation

data source Executable classes *not* requiring any other resources other than the parameters are called the *data source*. That is to say, an Executable class is a data source if its `run` method overriding `Executable.run` does *not* taking any arguments. Example:

`Loader`(*args, **kwds)	*Data source* loaded from disk.

data sink Executable classes requiring other resource are called the *data sink*. That is to say, an Executable class is a data sink if its `run` method overriding `Executable.run` takes *at least one* argument. Example:

`Plotter`(*args, **kwds)	An `Assembler` subclass specialized for plotting.

app

application Executable classes which orchestrate other executable classes are called *application* or *app*. Note that an *app* is also a *data source* since it does not require additional data source. `compapp.apps` defines a few base classes for this purpose:

`Computer`(*args, **kwds)	Application base class.
`Memoizer`(*args, **kwds)	`Computer` with `HashDataStore`.

Taxonomy of executables

Various `Executable` subclasses provided by compapp can be understood well when kind of computations, the type of `datastore` and used plugins are compared.

Executables with `SubDataStore` require parent executable. It fits well with `Loader` since just loading data is useless. It also fits well with `Plotter` since it is a data sink, i.e., it needs data for plotting. Since `Plotter` needs

some external *data source*, it makes sense that it is not a subclass of `Computer`.

Executable	`datastore`	Computation
`Loader`	`SubDataStore`	*data source*
`Plotter`	`SubDataStore`	*data sink*
`Computer`	`DirectoryDataStore`	*app*
`Memoizer`	`HashDataStore`	*app*

CHAPTER 5

Quick reference

Core

compapp.core	
Parametric(*args, **kwds)	The basic parametrized class.
Parametric.params([nested, type])	Get parameters as a dict.
Parametric.paramnames([type])	List names of parameter for this class.
Parametric.defaultparams([nested, type])	Get default parameters as a dict.

Executables

Executable(*args, **kwds)	The base class supporting execution and plugin mechanism.
Executable.prepare()	*[to be extended]* Prepare for run/load; e.g., execute upstreams.
Executable.run(*args)	*[to be extended]* Do the actual simulation/analysis.
Executable.save()	*[to be extended]* Save the result manually.
Executable.load()	*[to be extended]* Load saved result manually.
Executable.finish()	*[to be extended]* Do anything to be done before exit.
Executable.execute(*args)	Execute this instance.

executables.Loader(*args, **kwds)	*Data source* loaded from disk.
executables.Plotter(*args, **kwds)	An Assembler subclass specialized for plotting.
apps	Application base classes.
apps.Computer(*args, **kwds)	Application base class.
apps.Computer.cli([args])	Run Command Line Interface of this class.
apps.Memoizer(*args, **kwds)	Computer with HashDataStore.

Plugins

`Plugin`(*args, **kwds)	Plugin base class.
`Plugin.prepare`()	*[to be extended]* For a task immediately *after* `Executable.prepare`.
`Plugin.pre_run`()	*[to be extended]* For a task immediately before `Executable.run`.
`Plugin.post_run`()	*[to be extended]* For a task immediately after `Executable.run`.
`Plugin.save`()	*[to be extended]* For a task immediately *after* `Executable.save`.
`Plugin.load`()	*[to be extended]* For a task immediately *before* `Executable.load`.
`Plugin.finish`()	*[to be extended]* For a task immediately *before* `Executable.finish`.

compapp.plugins

`DirectoryDataStore`(*args, **kwds)	Data-store using a directory.
`SubDataStore`(*args, **kwds)	Data-store using sub-paths of parent data-store.
`HashDataStore`(*args, **kwds)	Automatically allocated data-store based on hash of parameter.
`MetaStore`(*args, **kwds)	
`Logger`(*args, **kwds)	Interface to pre-configured `logging.Logger`.
`Debug`(*args, **kwds)	Debug helper plugin.
`Figure`(*args, **kwds)	A wrapper around `matplotlib.pyplot.figure`.
`AutoUpstreams`(*args, **kwds)	Automatically execute upstreams.
`DumpResults`(*args, **kwds)	Automatically save owner's results.
`DumpParameters`(*args, **kwds)	Dump parameters used for its owner.
`RecordVCS`(*args, **kwds)	Record VCS revision automatically.
`RecordTiming`(*args, **kwds)	Record timing information.
`RecordProgramInfo`(*args, **kwds)	
`RecordSysInfo`(*args, **kwds)	

Descriptors

`OfType`(*classes, **kwds)	Attribute accepting only certain type(s) of value.
`Required`([desc])	Attributes required to be set before `Executable.run`.
`List`([trait, type, cast])	Attribute accepting only list with certain traits.
`Dict`([key, value, type, cast])	Attribute accepting only dict with certain traits.
`Optional`(*classes, **kwds)	Optional parameter.
`Choice`(default, *choices, **kwds)	Attribute accepting only one of the specified value.
`Or`(*traits, **kwds)	Use one of the specified traits.
`Link`(path[, adapter])	"Link" parameter.
`Root`(**kwds)	An alias of `Link('')`.
`Delegate`(**kwds)	Delegate parameter to its owner.
`MyName`([default, isparam])	

Continued on next page

Table 5.6 – continued from previous page

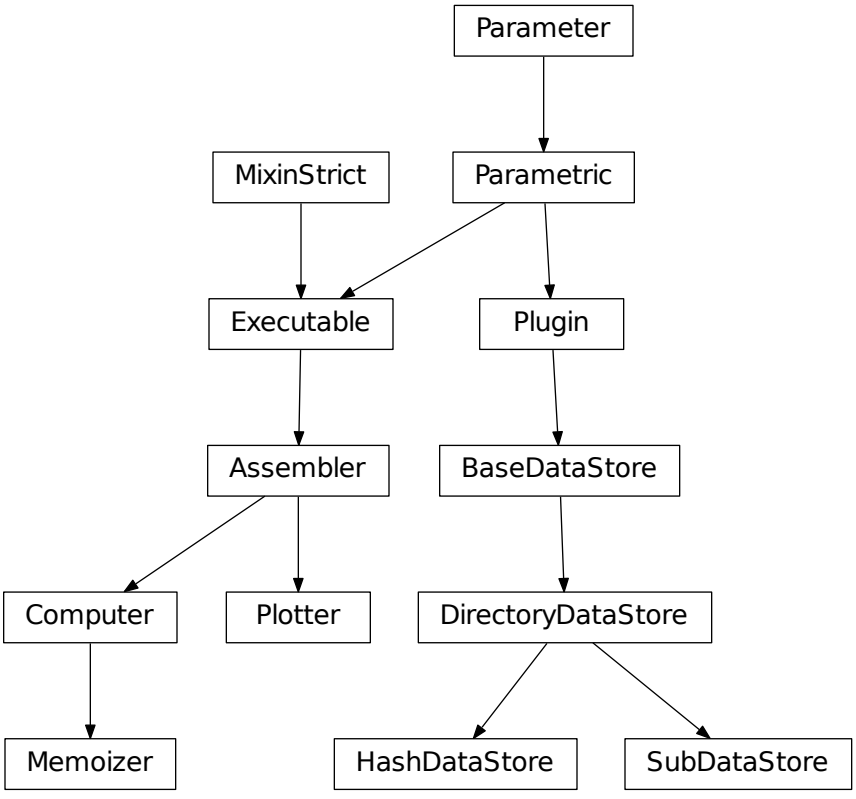
`OwnerName`([default, isparam])	
`Constant`(value)	Convenient descriptor for declaring non-parametric property.
`dynamic_class`(path[, prefix])	Dynamic class loading helper.
`ClassPath`(default[, prefix])	Class path descriptor that imports specified class on change.
`ClassPlaceholder`(cpath, **kwds)	Placeholder for an instance of the class specified by `ClassPath`.

Utilities

`setup_interactive`()	Configure compapp for REPL (e.g., IPython).

CHAPTER 6

Inheritance diagram



CHAPTER 7

Glossaries

datastore Datastore is the directory to put your simulation and analysis results. compapp may support more advanced data storage (e.g., data bases) in the future. See also `datastore` property.

nested class

owner class

owner app Schematically,:

```
class SubSimulator:
    pass

class MySimulator:          # owns SubSimulator
    sub = SubSimulator      # nests in SubSimulator

class MyApp:                # owns MySimulator
    sim = MySimulator       # nests in MyApp
```

In the above example:

- `MyApp` is the owner class of `MySimulator`.
- `MySimulator` is the owner class of `SubSimulator`.

In turn:

- `SubSimulator` is a nested class of `MySimulator`.
- `MySimulator` is a nested class of `MyApp`.

An owner class happened to be a subclass of `Computer` is called an *owner app*.

(Side note: the term *owner* is from the interface of Python descriptor; the `object.__get__` method receives the owner class as its last argument.)

TBE

to be extended Methods and properties marked as *to be extended* or *TBE* may be overridden (extended) by user-defined subclasses to implement certain functionalities. Note that the override is completely optional as oppose

to abstract methods and properties which are required to be overridden by subclasses. In Python code, docstrings for such methods and properties are prefixed with `|TO BE EXTENDED|`.

composition over inheritance See: Composition over inheritance - Wikipedia

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index