
compapp Documentation

Release 0.1.0.dev1

Takafumi Arakaki

May 30, 2018

Contents

1	What is compapp?	3
1.1	Automatic data directory naming, creation and management	3
1.2	Parameter management	3
1.3	Automatic type-check and value-check for properties (traits)	4
1.4	Linking properties	4
1.5	Hooks	4
1.6	Plugins	5
2	Tutorial	7
2.1	Plotting	8
2.2	Composition of computations	8
2.3	Dynamic loading	9
2.4	Trying out multiple parameters (in parallel)	10
3	Examples	11
3.1	Example: <code>compapp.samples.named_figure</code>	11
3.2	Example: <code>compapp.samples.pluggable_plotter_exec</code>	12
3.3	Example: <code>compapp.samples.pluggable_plotter_link</code>	15
3.4	Example: <code>compapp.samples.simple_plots</code>	15
3.5	Old examples	18
4	Executables	21
4.1	Kinds of computation	21
4.2	Taxonomy of executables	21
5	Quick reference	23
5.1	Core	23
5.2	Executables	23
5.3	Plugins	24
5.4	Descriptors	24
5.5	Utilities	25
6	Inheritance diagram	27
7	Glossaries	29
8	compapp package	31

8.1	Subpackages	31
8.2	Submodules	89
8.3	Module contents	113
9	compapp	115
10	Indices and tables	117
	Python Module Index	119

Contents:

What is compapp?

1.1 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.

<i>Computer & Executable subclasses</i>	<i>datastore</i> property	Behavior
<i>Computer</i>	<i>DirectoryDataStore</i>	Simulation is run with a specified data directory in which simulation parameters and results are saved. <i>Nested classes</i> such as <i>Plotter</i> and other nested <i>Computers</i> may use sub-paths.
<i>nested Computer</i>	<i>DirectoryDataStore</i>	If a <i>Computer</i> subclass nests in some <i>owner app</i> , <i>DirectoryDataStore</i> automatically allocates sub-directory under the directory used by the <i>owner app</i> .
<i>Plotter, Loader</i>	<i>SubDataStore</i>	Use files under the directory of the <i>owner app</i> .
<i>Memoizer</i>	<i>HashDataStore</i>	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 <i>Memoizer</i> 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.

1.2 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.

1.3 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.

1.4 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 easy¹) to share properties between parent and sub classes. compapp provides various *linking properties* (*Link*, *Delegate*, etc.) which can refer to properties of other objects.

1.5 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 *loaded*.

See also: [API Reference](#)

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

1.6 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*:

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

2.1 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.

2.2 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):
```

(continues on next page)

(continued from previous page)

```

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)

```

2.3 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)
```

2.4 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"]:level=(START,[ STOP[, STEP]])' \
    '--builder.linspaces["PATH.TO.A.PARAM"]:level=(START,[ STOP[, STEP]])' \
    '--builder.logspaces["PATH.TO.A.PARAM"]:level=(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)
```

3.1 Example: `compapp.samples.named_figure`

```
import numpy

from compapp.apps import Computer

class MyApp(Computer):

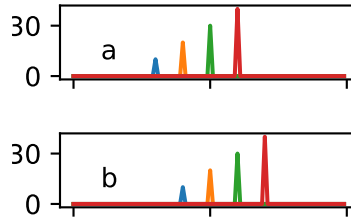
    def run(self):
        names = ['a', 'b', 'c']

        for name in names:
            fig = self.figure(figsize=(2, 0.5), name=name)
            fig.add_subplot(111)

            for j in range(1, 5):
                for i, name in enumerate(names):
                    x = numpy.zeros(100)
                    x[(i + j) * 10 + 20] = j * 10
                    self.figure[name].axes[0].plot(x)

            for name in names:
                ax = self.figure[name].axes[0]
                ax.set_yticks([0, 30])
                ax.text(10, 10, name)

if __name__ == '__main__':
    MyApp().execute()
```



3.2 Example: compapp.samples.pluggable_plotter_exec

```
import numpy

from compapp import Computer, Plotter

class HistPlotter(Plotter):
    bins = 100

    def run(self, x):
        _, ax = self.figure.subplots()
        ax.hist(x, **self.params())

class CumHistPlotter(HistPlotter):
    cumulative = True

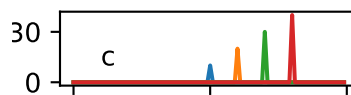
class MySimulator(Computer):
    samples = 1000

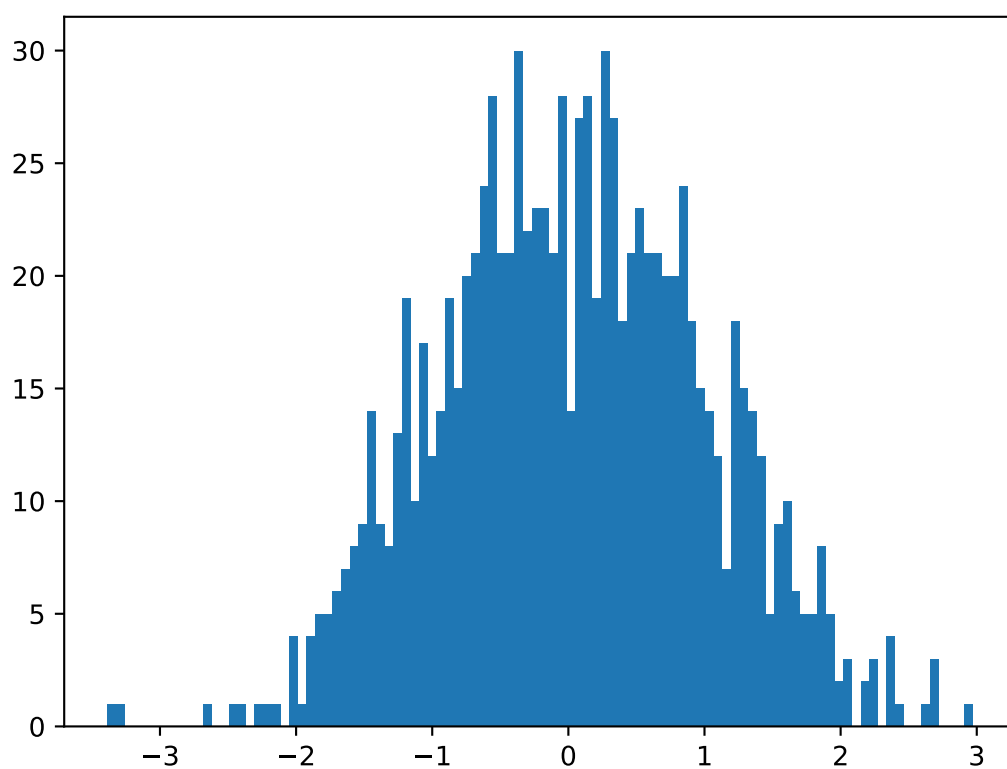
    def run(self):
        self.results.data = numpy.random.randn(self.samples)

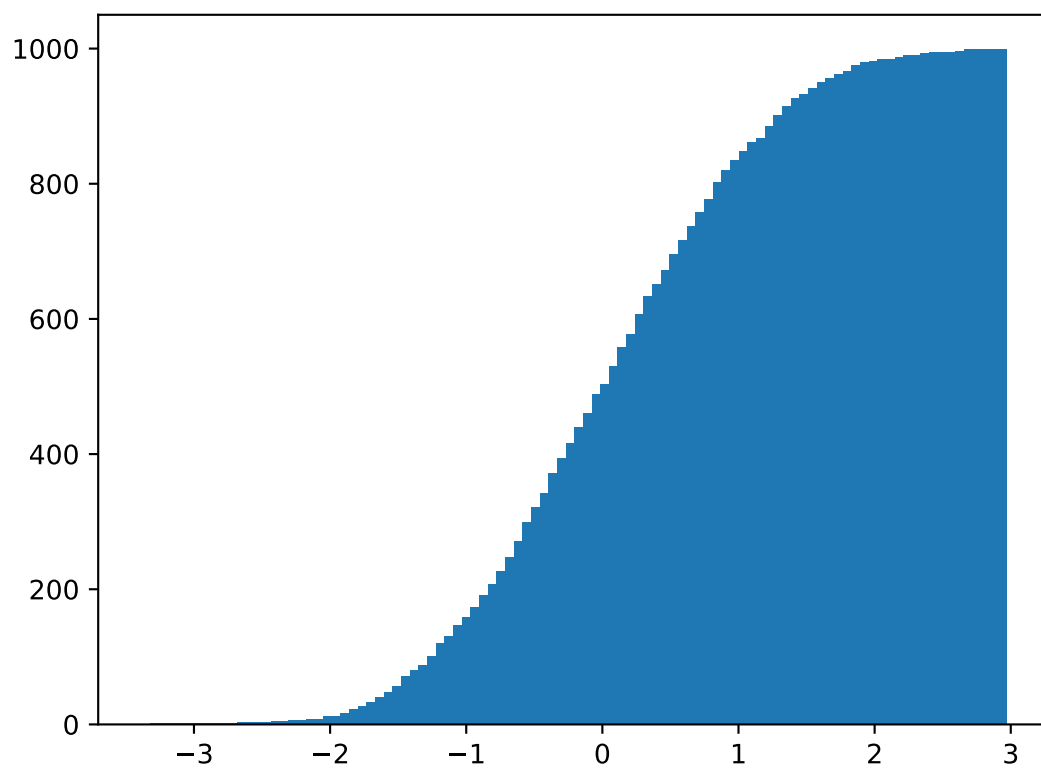
class MyApp(Computer):
    sim = MySimulator
    density = HistPlotter
    cumdist = CumHistPlotter

    def run(self):
        self.sim.execute()
        self.density.execute(self.sim.results.data)
        self.cumdist.execute(self.sim.results.data)

if __name__ == '__main__':
    MyApp().execute()
```







3.3 Example: compapp.samples.pluggable_plotter_link

```
import numpy

from compapp import Computer, Plotter, Link, AutoUpstreams

class HistPlotter(Plotter):
    bins = 100
    data = Link('..sim.results.data')

    def run(self):
        _, ax = self.figure.subplots()
        ax.hist(self.data, **self.params())

class CumHistPlotter(HistPlotter):
    cumulative = True

class MySimulator(Computer):
    samples = 1000

    def run(self):
        self.results.data = numpy.random.randn(self.samples)

class MyApp(Computer):
    autoupstreams = AutoUpstreams
    sim = MySimulator
    density = HistPlotter
    cumdist = CumHistPlotter

if __name__ == '__main__':
    MyApp().execute()
```

3.4 Example: compapp.samples.simple_plots

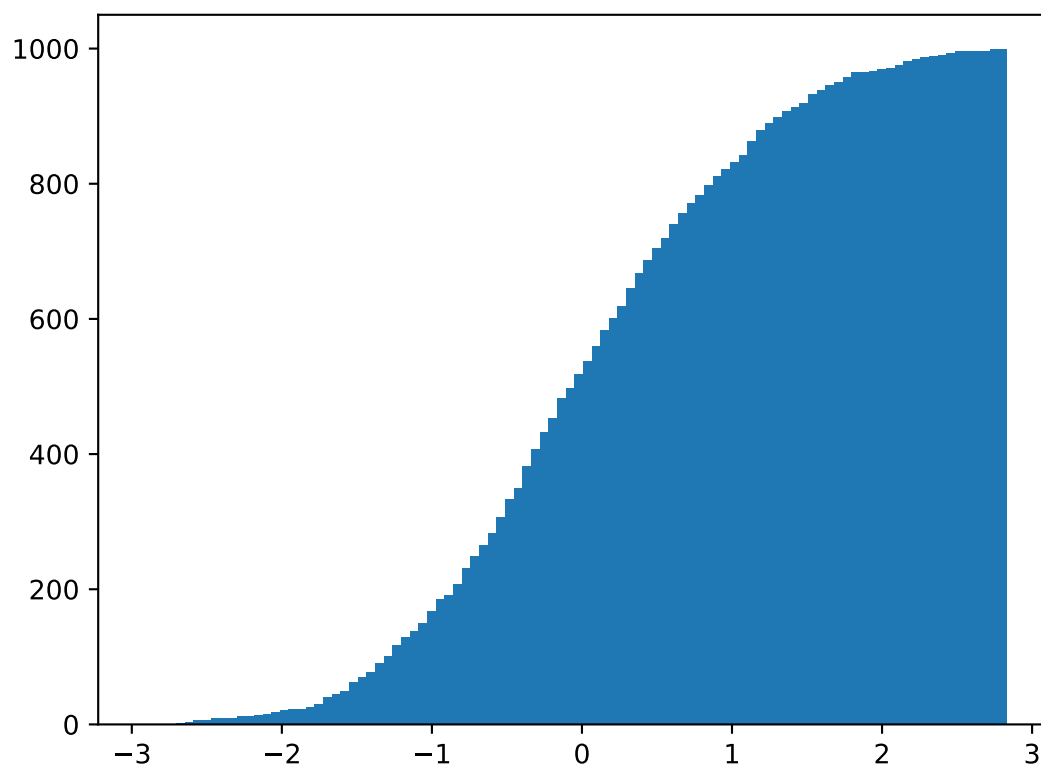
```
from compapp.apps import Computer

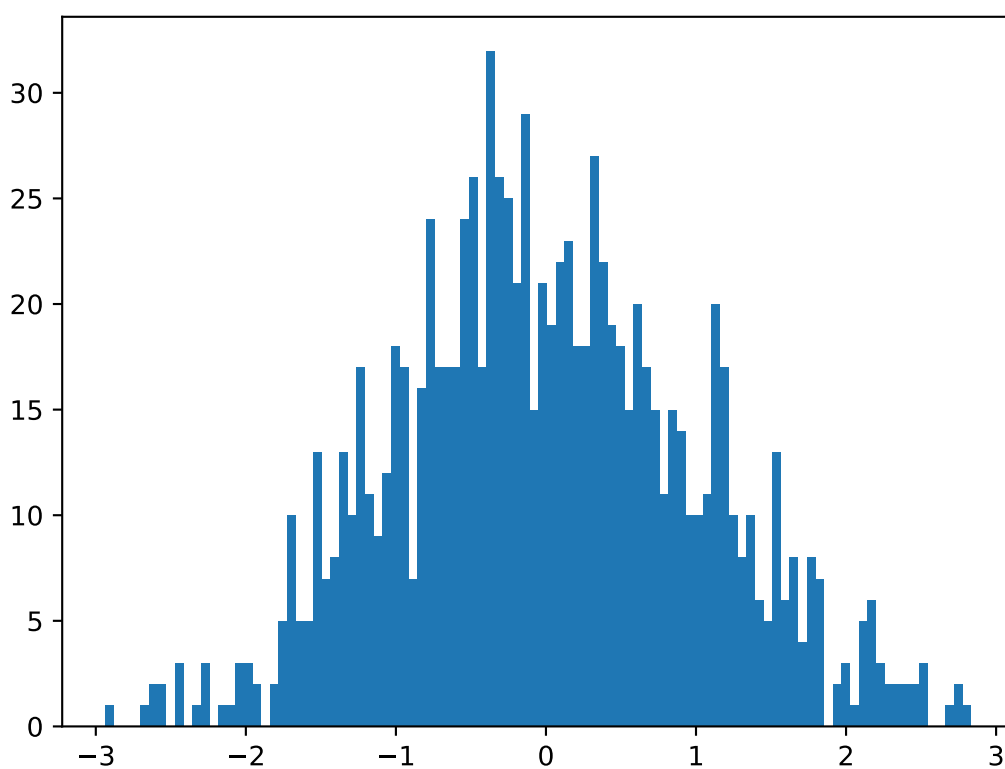
class MyApp(Computer):

    def run(self):
        fig1 = self.figure()
        ax1 = fig1.add_subplot(111)
        ax1.plot([x ** 2 for x in range(100)])

        fig2, (ax21, ax22) = self.figure.subplots(2)
        ax21.plot([0, 2, 1, 3, 5])
        ax22.plot([0, 1, 5, 3, 0])
```

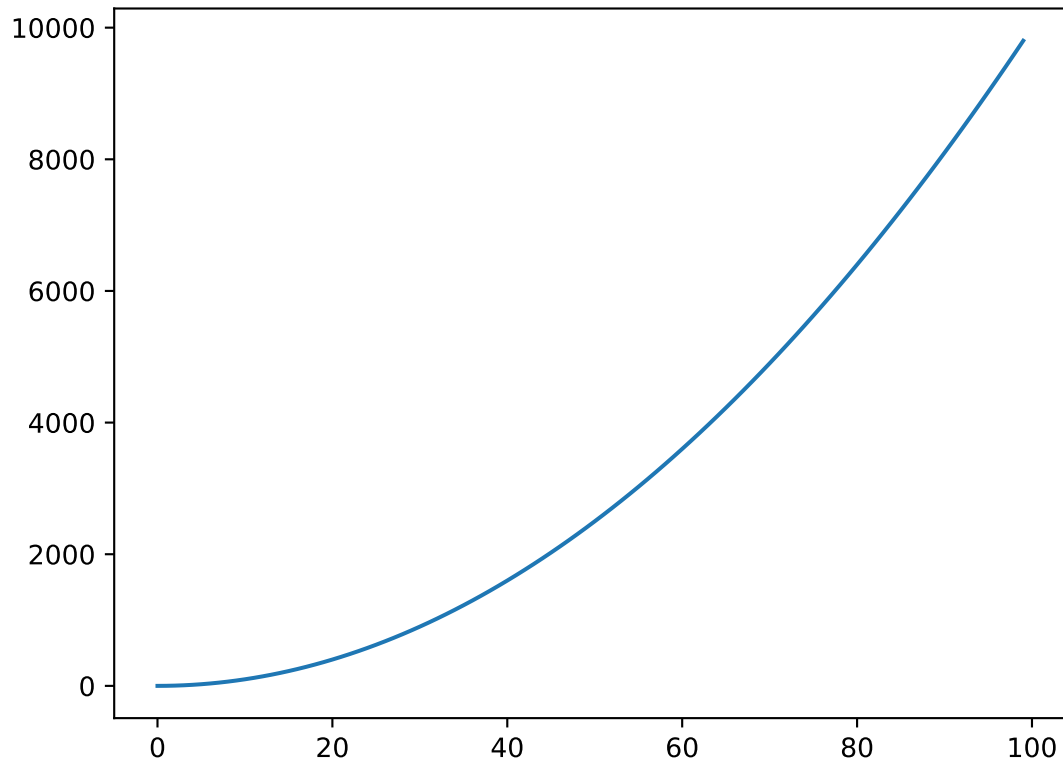
(continues on next page)





(continued from previous page)

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

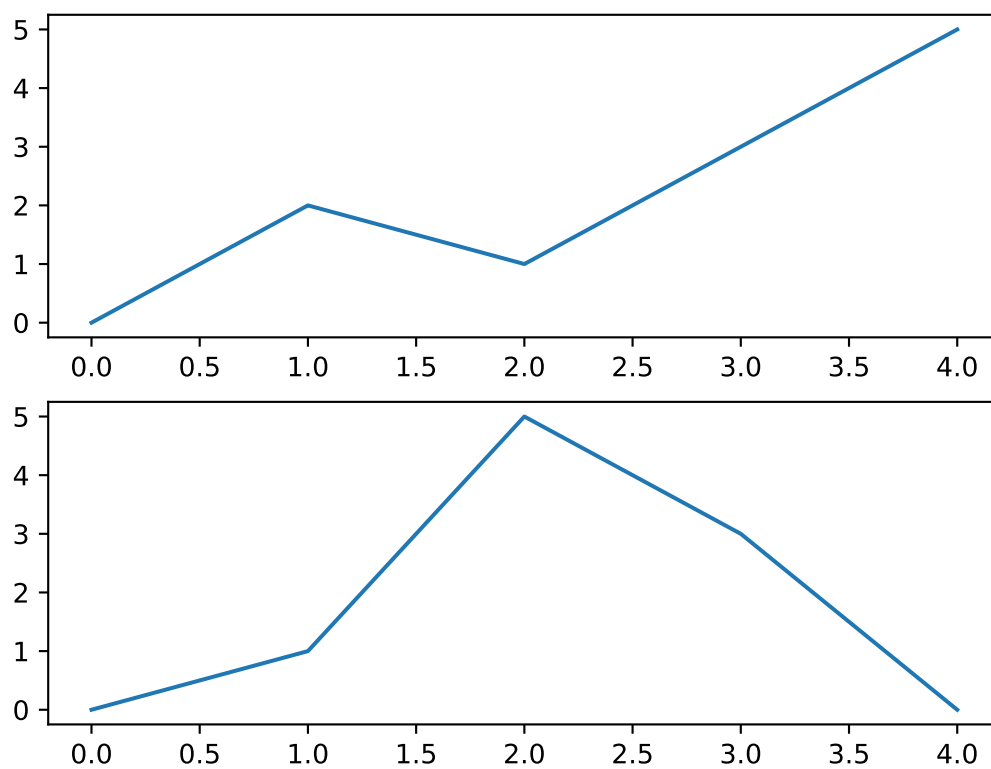


3.5 Old examples

3.5.1 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
```

(continues on next page)



(continued from previous page)

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

Run:

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

Result:

```
OUT/results.json
```

3.5.2 Simple plotting example

```
from compapp import Computer  
  
class PlottingApp(Computer):  
  
    def run(self):  
        fig = self.figure()  
        ax = fig.add_axes([1, 1, 1])  
        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
```


4.1 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:

<code>Loader(*args, **kwds)</code>	<i>Data source</i> 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:

<code>Plotter(*args, **kwds)</code>	An <i>Assembler</i> 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:

<code>Computer(*args, **kwds)</code>	Application base class.
<code>Memoizer(*args, **kwds)</code>	<i>Computer</i> with <i>HashDataStore</i> .

4.2 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
<i>Loader</i>	<i>SubDataStore</i>	<i>data source</i>
<i>Plotter</i>	<i>SubDataStore</i>	<i>data sink</i>
<i>Computer</i>	<i>DirectoryDataStore</i>	<i>app</i>
<i>Memoizer</i>	<i>HashDataStore</i>	<i>app</i>

5.1 Core

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

5.2 Executables

<i>Executable(*args, **kws)</i>	The base class supporting execution and plugin mechanism.
<i>Executable.prepare()</i>	<i>[to be extended]</i> Prepare for run/load; e.g., execute upstreams.
<i>Executable.run(*args)</i>	<i>[to be extended]</i> Do the actual simulation/analysis.
<i>Executable.save()</i>	<i>[to be extended]</i> Save the result manually.
<i>Executable.load()</i>	<i>[to be extended]</i> Load saved result manually.
<i>Executable.finish()</i>	<i>[to be extended]</i> Do anything to be done before exit.
<i>Executable.execute(*args)</i>	Execute this instance.
<i>executables.Loader(*args, **kws)</i>	<i>Data source</i> loaded from disk.
<i>executables.Plotter(*args, **kws)</i>	An <i>Assembler</i> subclass specialized for plotting.
<i>apps</i>	Application base classes.
<i>apps.Computer(*args, **kws)</i>	Application base class.
<i>apps.Computer.cli([args])</i>	Run Command Line Interface of this class.
<i>apps.Memoizer(*args, **kws)</i>	Computer with <i>HashDataStore</i> .

5.3 Plugins

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

5.3.1 compapp.plugins

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

5.4 Descriptors

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

Continued on next page

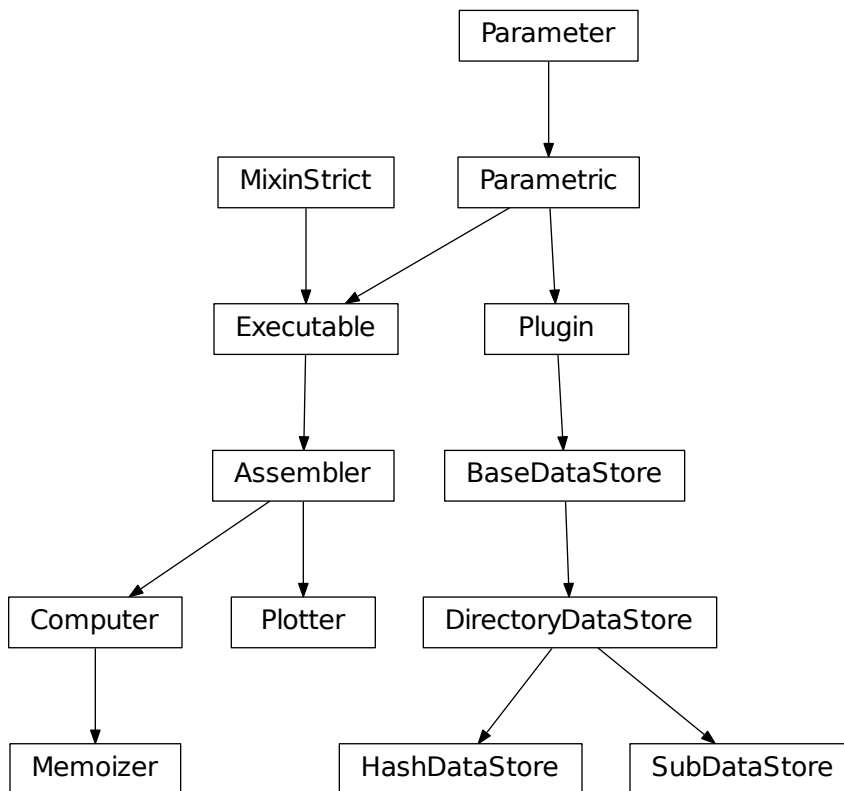
Table 6 – continued from previous page

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

5.5 Utilities

<i>setup_interactive</i> ()	Configure compapp for REPL (e.g., IPython).
-----------------------------	---

Inheritance diagram



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

owner 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*. An instance of a owner class is simply called an *owner*.

(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](#)

8.1 Subpackages

8.1.1 compapp.descriptors package

Subpackages

compapp.descriptors.tests package

Submodules

compapp.descriptors.tests.test_complex module

class compapp.descriptors.tests.test_complex.**Pluggy**(*args, **kws)
Bases: *compapp.core.Parametric*

fallback

Use one of the specified traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     attr = Or(Choice(*'abc'), OfType(int))
...
>>> mp = MyParametric()
>>> mp.attr = 'a'
>>> mp.attr = 1
>>> mp.attr = 1.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: None of the traits are matched to: 1.0
```

Or can be combined with *Link*-like descriptors:

```
>>> from compapp.descriptors import Delegate
>>> class MyRoot (Parametric):
...     attr = Choice(*'abc', nodelault=True)
...
...     class nested (Parametric):
...         attr = Or(OfType(int), Delegate())
...
>>> par = MyRoot()
>>> par.nested.attr
Traceback (most recent call last):
...
AttributeError: 'nested' object has no attribute 'attr'
```

If `par.nested.attr` is not specified (i.e., `OfType(int)` is not in action), accessing to `par.nested.attr` invokes *Delegate* which tries to access `par.attr`. In the above example, `par.attr` was not set so the *AttributeError* was raised. Setting `par.attr` makes `par.nested.attr` accessible:

```
>>> par.attr = 'b'
>>> par.nested.attr
'b'
```

Specifying `par.nested.attr` directly invokes *OfType*:

```
>>> par.nested.attr = 'c'
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 'c'
>>> par.nested.attr = 496
>>> par.nested.attr
496
```

```
__module__ = 'compapp.descriptors.tests.test_complex'

class compapp.descriptors.tests.test_complex.Nestable (*args, **kws)
    Bases: compapp.core.Parametric

    pluggy
        alias of Pluggy

    __module__ = 'compapp.descriptors.tests.test_complex'

class compapp.descriptors.tests.test_complex.Root (*args, **kws)
    Bases: compapp.descriptors.tests.test_complex.Nestable

class sub (*args, **kws)
    Bases: compapp.descriptors.tests.test_complex.Nestable

class sub (*args, **kws)
    Bases: compapp.descriptors.tests.test_complex.Nestable

class sub (*args, **kws)
    Bases: compapp.descriptors.tests.test_complex.Nestable
```

```

class sub(*args, **kws)
    Bases: compapp.descriptors.tests.test_complex.Nestable
    __module__ = 'compapp.descriptors.tests.test_complex'
    __module__ = 'compapp.descriptors.tests.test_complex'
    __module__ = 'compapp.descriptors.tests.test_complex'
    __module__ = 'compapp.descriptors.tests.test_complex'
    __module__ = 'compapp.descriptors.tests.test_complex'
compapp.descriptors.tests.test_complex.assert_fallbacks_are(app, what)
compapp.descriptors.tests.test_complex.dassert_fallbacks_are(obj, what)
compapp.descriptors.tests.test_complex.test_propagation()

```

compapp.descriptors.tests.test_dynamic_class module

compapp.descriptors.tests.test_or module

```

class compapp.descriptors.tests.test_or.ParOrDelegate(*args, **kws)
    Bases: compapp.core.Parametric
    attr = 1
    class nested(*args, **kws)
        Bases: compapp.core.Parametric
        attr
            Use one of the specified traits.

```

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     attr = Or(Choice(*'abc'), OfType(int))
...
>>> mp = MyParametric()
>>> mp.attr = 'a'
>>> mp.attr = 1
>>> mp.attr = 1.0
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 1.0

```

Or can be combined with *Link*-like descriptors:

```

>>> from compapp.descriptors import Delegate
>>> class MyRoot(Parametric):
...     attr = Choice(*'abc', nodefault=True)
...
...     class nested(Parametric):
...         attr = Or(OfType(int), Delegate())
...
>>> par = MyRoot()

```

(continues on next page)

(continued from previous page)

```
>>> par.nested.attr
Traceback (most recent call last):
...
AttributeError: 'nested' object has no attribute 'attr'
```

If `par.nested.attr` is not specified (i.e., `OfType(int)` is not in action), accessing to `par.nested.attr` invokes *Delegate* which tries to access `par.attr`. In the above example, `par.attr` was not set so the `AttributeError` was raised. Setting `par.attr` makes `par.nested.attr` accessible:

```
>>> par.attr = 'b'
>>> par.nested.attr
'b'
```

Specifying `par.nested.attr` directly invokes *OfType*:

```
>>> par.nested.attr = 'c'
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 'c'
>>> par.nested.attr = 496
>>> par.nested.attr
496
```

```
__module__ = 'compapp.descriptors.tests.test_or'
__module__ = 'compapp.descriptors.tests.test_or'
compapp.descriptors.tests.test_or.test_key_propagation()
compapp.descriptors.tests.test_or.test_myname_propagation()
compapp.descriptors.tests.test_or.test_propagated_myname()
compapp.descriptors.tests.test_or.test_function()
```

compapp.descriptors.tests.test_params module

```
compapp.descriptors.tests.test_params.test_dict_wo_default()
compapp.descriptors.tests.test_params.test_list_wo_default()
class compapp.descriptors.tests.test_params.TestDictWithDefault
    Bases: object
    class MyApp(*args, **kws)
        Bases: compapp.core.Parametric
        x
        Attribute accepting only dict with certain traits.
```

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
```

(continues on next page)

(continued from previous page)

```

...     strint = Dict(str, int)
...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str

```

```

__module__ = 'compapp.descriptors.tests.test_params'

test()

__dict__ = dict_proxy({'__module__': 'compapp.descriptors.tests.test_params', '__dict__':
__module__ = 'compapp.descriptors.tests.test_params'
__weakref__
    list of weak references to the object (if defined)

class compapp.descriptors.tests.test_params.TestOrDefault
    Bases: compapp.descriptors.tests.test_params.TestDictWithDefault

class MyApp(*args, **kws)
    Bases: compapp.core.Parametric

    x
        Use one of the specified traits.

```

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     attr = Or(Choice(*'abc'), OfType(int))
...
>>> mp = MyParametric()
>>> mp.attr = 'a'
>>> mp.attr = 1
>>> mp.attr = 1.0
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 1.0

```

Or can be combined with *Link*-like descriptors:

```

>>> from compapp.descriptors import Delegate
>>> class MyRoot(Parametric):
...     attr = Choice(*'abc', nodefault=True)
...
...     class nested(Parametric):

```

(continues on next page)

(continued from previous page)

```
...         attr = Or(OfType(int), Delegate())
...
>>> par = MyRoot()
>>> par.nested.attr
Traceback (most recent call last):
...
AttributeError: 'nested' object has no attribute 'attr'
```

If `par.nested.attr` is not specified (i.e., `OfType(int)` is not in action), accessing to `par.nested.attr` invokes *Delegate* which tries to access `par.attr`. In the above example, `par.attr` was not set so the *AttributeError* was raised. Setting `par.attr` makes `par.nested.attr` accessible:

```
>>> par.attr = 'b'
>>> par.nested.attr
'b'
```

Specifying `par.nested.attr` directly invokes *OfType*:

```
>>> par.nested.attr = 'c'
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 'c'
>>> par.nested.attr = 496
>>> par.nested.attr
496
```

```
__module__ = 'compapp.descriptors.tests.test_params'
__module__ = 'compapp.descriptors.tests.test_params'
compapp.descriptors.tests.test_params.test_link_default()
```

Module contents

Submodules

compapp.descriptors.dynamic_class module

Descriptors for dynamic class loading

Implementation details

dynamic_class works by defining a pair of descriptors for holding “class path” (*ClassPath*) and the instance of the class (*ClassPlaceholder*). Those descriptors make sure that they are consistent; changing *ClassPath* “sets” the class of the objects at *ClassPlaceholder* and setting the instance directly at *ClassPlaceholder* changes *ClassPath*. Accessing the value at *ClassPlaceholder* when it is inconsistent with *ClassPath* raises a *ValueError*.

class compapp.descriptors.dynamic_class.**ClassPath** (*default*, *prefix=None*)

Bases: *compapp.core.DataDescriptor*

Class path descriptor that imports specified class on change.


```

__init__(default, prefix=None)
    x.__init__(...) initializes x; see help(type(x)) for signature
verify(obj, value, myname=None)
getclass(obj)
__module__ = 'compapp.descriptors.dynamic_class'

```

class compapp.descriptors.dynamic_class.**ClassPlaceholder**(*cpath*, ***kws*)

Bases: [compapp.core.DataDescriptor](#)

Placeholder for an instance of the class specified by [ClassPath](#).

The actual instantiation process is delayed until it is accessed (i.e., `__get__` method is called).

```

__init__(cpath, **kws)
    x.__init__(...) initializes x; see help(type(x)) for signature
get(obj)
__set__(obj, value)
__module__ = 'compapp.descriptors.dynamic_class'

```

compapp.descriptors.dynamic_class.**dynamic_class**(*path*, *prefix=None*, ***kws*)

Dynamic class loading helper.

Parameters

- **path** (*str* or *type*) – The path to default class. The actual class can also be passed.
- **prefix** (*str* or *NoneType*) – The module path at which the relative path is resolved.
- **default** (*dict*) – The default parameter that is passed to the `__init__` method of the class at *path*.
- **isparam** (*bool*) – If *False*, do not include the class in parameter.

Examples

```

>>> from compapp import Parametric, dynamic_class
>>> class MyApp(Parametric):
...     obj, path = dynamic_class('.ClassA', prefix=__name__, default={})
...
>>> class ClassA(object):
...     def __init__(self, params):
...         self.params = params
...
>>> class ClassB(ClassA):
...     pass
...

```

An instance of a class is automatically loaded at `app.obj`:

```

>>> app = MyApp()
>>> app.obj
<{module name}.ClassA object at 0x...>
>>> app.obj.params
{}

```

The class to be loaded can be changed dynamically by specifying the `path` attribute/parameter. The path can be full or relative, if the `prefix` argument is specified.

```
>>> app2 = MyApp(obj={'a': 1}, path='.ClassB')
>>> app2.obj
<{module name}.ClassB object at 0x...>
>>> app2.obj.params
{'a': 1}
```

The object can also be set manually. In that case, the corresponding path would be set to the full dotted path.

```
>>> app3 = MyApp()
>>> app3.obj = ClassB({})
>>> app3.path
'{module name}.ClassB'
```

compapp.descriptors.links module

`compapp.descriptors.links.countprefix` (*string*, *prefix*)

class `compapp.descriptors.links.Link` (*path*, *adapter=None*, ***kwds*)

Bases: `compapp.core.Descriptor`

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
```

(continues on next page)

(continued from previous page)

```

>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```

__init__(path, adapter=None, **kws)
    x.__init__(...) initializes x; see help(type(x)) for signature

```

get (*obj*)

```

__module__ = 'compapp.descriptors.links'

```

```

class compapp.descriptors.links.Root(**kws)
    Bases: compapp.descriptors.links.Link

```

An alias of Link('').

```

__init__(**kws)
    x.__init__(...) initializes x; see help(type(x)) for signature

```

```

__module__ = 'compapp.descriptors.links'

```

```

class compapp.descriptors.links.Delegate(**kws)
    Bases: compapp.descriptors.links.Link

```

Delegate parameter to its owner.

x = Delegate() is equivalent to x = Link('..x').

Examples

```

>>> from compapp.core import Parametric
>>> class Parent(Parametric):
...
...     class nest1(Parametric):
...         sampling_period = Delegate()
...
...     class nest2(Parametric):
...         sampling_period = Delegate()

```

(continues on next page)

(continued from previous page)

```
...
...     sampling_period = 10.0
...
>>> par = Parent()
>>> par.sampling_period
10.0
>>> par.nest1.sampling_period
10.0
>>> par.nest2.sampling_period
10.0
>>> par.sampling_period = 20.0
>>> par.nest1.sampling_period
20.0
```

```
__init__(**kws)
    x.__init__(...) initializes x; see help(type(x)) for signature
```

```
get(obj)
```

```
__module__ = 'compapp.descriptors.links'
```

```
class compapp.descriptors.links.MyName (default=Unspecified, isparam=Unspecified)
    Bases: compapp.core.Descriptor
```

```
get(obj)
```

```
__module__ = 'compapp.descriptors.links'
```

```
class compapp.descriptors.links.OwnerName (default=Unspecified, isparam=Unspecified)
    Bases: compapp.core.Descriptor
```

```
get(obj)
```

```
__module__ = 'compapp.descriptors.links'
```

compapp.descriptors.misc module

```
class compapp.descriptors.misc.Constant (value)
    Bases: compapp.core.Descriptor
```

Convenient descriptor for declaring non-parametric property.

```
>>> from compapp import Parametric, Constant
>>> class MyApp(Parametric):
...     c = Constant(1)
>>> app = MyApp()
>>> app.c
1
>>> app.c = 2
Traceback (most recent call last):
...
TypeError: can't set attributes <...MyApp ...>.c
>>> app.c
1
```

```
__init__(value)
    x.__init__(...) initializes x; see help(type(x)) for signature
```

```
get(_)
```

```
__set__(obj, value)
__module__ = 'compapp.descriptors.misc'
```

compapp.descriptors.traits module

```
compapp.descriptors.traits.tupleoftypes(t)
compapp.descriptors.traits.skip_non_str(func)
```

class compapp.descriptors.traits.OfType(**classes*, ***kwds*)

Bases: [compapp.core.DataDescriptor](#)

Attribute accepting only certain type(s) of value.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     i = OfType(int)
...
>>> MyParametric(i='a')
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int: got 'a' of type str
```

OfType can take multiple classes:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, float, str)
...
>>> mp = MyParametric()
>>> mp.i = 'a'
>>> mp.i = 1j
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int, float or str: got 1j
of type complex
```

It is an error to access unset an *OfType* attribute:

```
>>> mp = MyParametric()
>>> mp.i
Traceback (most recent call last):
...
AttributeError: 'MyParametric' object has no attribute 'i'
```

OfType can take *default* argument:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, default=0)
...
>>> mp = MyParametric()
>>> mp.i
0
```

Castable values are cast automatically:

```
>>> class MyParametric(Parametric):
...     x = OfType(float)
...
>>> mp = MyParametric()
>>> mp.x = 1                                # assigning an int...
>>> assert isinstance(mp.x, float)         # ... cast to a float
>>> import numpy
>>> mp.x = numpy.float16(2.0)              # assigning a numpy float...
>>> assert isinstance(mp.x, float)         # ... cast to a float
```

__init__ (*classes, **kws)
x.__init__(...) initializes x; see help(type(x)) for signature

allowed

verify (obj, value, myname=None)

get (obj)

parse (value)

__module__ = 'compapp.descriptors.traits'

class compapp.descriptors.traits.**Required** (desc=None)

Bases: [compapp.core.DataDescriptor](#)

Attributes required to be set before [Executable.run](#).

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     i = Required()
...
>>> mp = MyParametric()
>>> has_required_attributes(mp)
False
>>> mp.i = 1
>>> has_required_attributes(mp)
True
```

```
>>> class MyParametric(Parametric):
...     i = Required(OfType(int))
...
>>> mp = MyParametric()
>>> has_required_attributes(mp)
False
>>> mp.i = '1'
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int: got '1' of type str
>>> mp.i = 1
>>> has_required_attributes(mp)
True
```

__init__ (desc=None)
x.__init__(...) initializes x; see help(type(x)) for signature

```
__module__ = 'compapp.descriptors.traits'
```

compapp.descriptors.traits.has_required_attributes(*obj*)

compapp.descriptors.traits.asdesc(*trait*)

class compapp.descriptors.traits.List(*trait=None, type=<type 'list'>, cast=None, **kws*)

Bases: [compapp.descriptors.traits.OfType](#)

Attribute accepting only list with certain traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anylist = List()
...     intlist = List(int)
...     castlist = List(int, cast=tuple)
...
>>> mp = MyParametric()
>>> mp.anylist = [1]
>>> mp.anylist = ['a']
>>> mp.intlist = [1]
>>> mp.intlist = [0, 1, '2']
Traceback (most recent call last):
...
ValueError: MyParametric.intlist[2] only accepts type of int: got '2'
of type str
```

```
>>> mp.intlist = (1,)
Traceback (most recent call last):
...
ValueError: MyParametric.intlist only accepts type of list: got (1,)
of type tuple
>>> mp.castlist = (1,)
>>> mp.castlist
[1]
```

```
__init__(trait=None, type=<type 'list'>, cast=None, **kws)
    x.__init__(...) initializes x; see help(type(x)) for signature
```

verify(*obj, value, myname=None*)

```
__module__ = 'compapp.descriptors.traits'
```

class compapp.descriptors.traits.Dict(*key=None, value=None, type=<type 'dict'>, cast=None, **kws*)

Bases: [compapp.descriptors.traits.OfType](#)

Attribute accepting only dict with certain traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
...     strint = Dict(str, int)
```

(continues on next page)

(continued from previous page)

```

...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str

```

__init__ (*key=None, value=None, type=<type 'dict'>, cast=None, **kwds*)
x.__init__(...) initializes *x*; see help(type(*x*)) for signature

verify (*obj, value, myname=None*)

__module__ = 'compapp.descriptors.traits'

class compapp.descriptors.traits.**Optional** (**classes, **kwds*)

Bases: *compapp.descriptors.traits.OfType*

Optional parameter.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     i = Optional(int)
...     j = Optional(int)
>>> sorted(MyParametric.paramnames())
['i', 'j']
>>> MyParametric().params()
{}
>>> MyParametric(i=1).params()
{'i': 1}
>>> MyParametric(j=2).params()
{'j': 2}
>>> assert MyParametric(i=1, j=2).params() == {'i': 1, 'j': 2}
>>> MyParametric(i='alpha')
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int: got 'alpha'
of type str

```

This is useful when writing Parametric interface to external library because you would like to avoid writing all default parameters in this case. A simple interface to `matplotlib.pyplot.hist` can be written as:

```

>>> class Hist(Parametric):
...
...     bins = Optional(int)
...     normed = Optional(bool)
...

```

(continues on next page)

(continued from previous page)

```

...     def plot(self, ax, x):
...         ax.hist(x, **self.params())
...
>>> from matplotlib import pyplot
>>> fig, ax = pyplot.subplots()
>>> Hist(bins=100, normed=True).plot(ax, range(100))
>>> pyplot.close(fig)

```

verify (obj, value, myname=None)

__module__ = 'compapp.descriptors.traits'

class compapp.descriptors.traits.Choice (default, *choices, **kwds)

Bases: [compapp.core.DataDescriptor](#)

Attribute accepting only one of the specified value.

Parameters

- **default** – Default value (see *nodefault*).
- ***choices** – Alternative values.
- **nodefault** (*bool*) – Treat *default* is just an alternative; i.e., do not “initialize” the attribute.
- ****kwds** – See: [DataDescriptor](#).

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     choice = Choice(1, 2, 'a')
...
>>> mp = MyParametric()
>>> mp.choice
1
>>> mp.choice = 2
>>> mp.choice = 'a'
>>> mp.choice = 'unknown'
Traceback (most recent call last):
...
ValueError: MyParametric.choice only accepts one of (1, 2, 'a'):
got 'unknown'

```

__init__ (default, *choices, **kwds)

x.**__init__**(...) initializes x; see `help(type(x))` for signature

verify (obj, value, myname=None)

parse (value)

__module__ = 'compapp.descriptors.traits'

class compapp.descriptors.traits.Or (*traits, **kwds)

Bases: [compapp.core.DataDescriptor](#)

Use one of the specified traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     attr = Or(Choice(*'abc'), OfType(int))
...
>>> mp = MyParametric()
>>> mp.attr = 'a'
>>> mp.attr = 1
>>> mp.attr = 1.0
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 1.0
```

Or can be combined with *Link*-like descriptors:

```
>>> from compapp.descriptors import Delegate
>>> class MyRoot(Parametric):
...     attr = Choice(*'abc', nodefault=True)
...
...     class nested(Parametric):
...         attr = Or(OfType(int), Delegate())
...
>>> par = MyRoot()
>>> par.nested.attr
Traceback (most recent call last):
...
AttributeError: 'nested' object has no attribute 'attr'
```

If `par.nested.attr` is not specified (i.e., `OfType(int)` is not in action), accessing to `par.nested.attr` invokes *Delegate* which tries to access `par.attr`. In the above example, `par.attr` was not set so the `AttributeError` was raised. Setting `par.attr` makes `par.nested.attr` accessible:

```
>>> par.attr = 'b'
>>> par.nested.attr
'b'
```

Specifying `par.nested.attr` directly invokes *OfType*:

```
>>> par.nested.attr = 'c'
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 'c'
>>> par.nested.attr = 496
>>> par.nested.attr
496
```

```
__init__(*traits, **kws)
    x.__init__(...) initializes x; see help(type(x)) for signature
```

key

```
__module__ = 'compapp.descriptors.traits'
```

myname

```
verify(obj, value, myname=None)
```

```
get(obj)
```

`parse (value)``default`

Module contents

8.1.2 compapp.plugins package

Subpackages

compapp.plugins.tests package

Submodules

compapp.plugins.tests.test_autoupstreams module

```

class compapp.plugins.tests.test_autoupstreams.AlwaysReady (*args, **kws)
    Bases: compapp.executables.Assembler

    run ()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.plugins.tests.test_autoupstreams'

class compapp.plugins.tests.test_autoupstreams.DependsOnAlwaysReady (*args,
                                                                    **kws)
    Bases: compapp.plugins.tests.test_autoupstreams.AlwaysReady

    alwaysready_done
        “Link” parameter.

        It’s like symbolic-link in the file system.

```

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             10 = Link('')
...             11 = Link('x')
...             12 = Link('..x')
...             13 = Link('.nest.x')
...             broken = Link('..broken')

```

(continues on next page)

(continued from previous page)

```
...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```
__module__ = 'compapp.plugins.tests.test_autoupstreams'
```

```
class compapp.plugins.tests.test_autoupstreams.DependsOnDepAR(*args, **kws)
```

```
Bases: compapp.plugins.tests.test_autoupstreams.AlwaysReady
```

depar_done

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
...     class nest(Parametric):
```

(continues on next page)

(continued from previous page)

```

...         x = 3
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```
__module__ = 'compapp.plugins.tests.test_autoupstreams'
```

```
class compapp.plugins.tests.test_autoupstreams.DependsonTwo(*args, **kws)
```

Bases: *compapp.plugins.tests.test_autoupstreams.DependsonDepAR*

deppar_done

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         10 = Link('')

```

(continues on next page)

(continued from previous page)

```

...     11 = Link('x')
...     12 = Link('..x')
...     13 = Link('.nest.x')
...     broken = Link('..broken')
...
...     class nest(Parametric):
...         x = 3
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```

__module__ = 'compapp.plugins.tests.test_autoupstreams'

class compapp.plugins.tests.test_autoupstreams.RootApp(*args, **kws)
    Bases: compapp.executables.Assembler

    autoupstreams
        alias of compapp.plugins.misc.AutoUpstreams

    alwaysready
        alias of AlwaysReady

    depar
        alias of DependsOnAlwaysReady

    depdepar
        alias of DependsOnDepAR

```

deptwo

alias of *DependsOnTwo*

__module__ = 'compapp.plugins.tests.test_autoupstreams'

compapp.plugins.tests.test_autoupstreams.test_execute()

compapp.plugins.tests.test_autoupstreams.test_is_runnable_simple()

compapp.plugins.tests.test_datastores module

class compapp.plugins.tests.test_datastores.WithStore(*args, **kws)

Bases: *compapp.interface.Executable*

datastore

alias of *compapp.plugins.datastores.DirectoryDataStore*

exists = None

run()

[to be extended] Do the actual simulation/analysis.

__module__ = 'compapp.plugins.tests.test_datastores'

class compapp.plugins.tests.test_datastores.CheckDir(*args, **kws)

Bases: *compapp.plugins.tests.test_datastores.WithStore*

class sub(*args, **kws)

Bases: *compapp.plugins.tests.test_datastores.WithStore*

class sub(*args, **kws)

Bases: *compapp.plugins.tests.test_datastores.WithStore*

class sub(*args, **kws)

Bases: *compapp.plugins.tests.test_datastores.WithStore*

__module__ = 'compapp.plugins.tests.test_datastores'

__module__ = 'compapp.plugins.tests.test_datastores'

__module__ = 'compapp.plugins.tests.test_datastores'

run()

[to be extended] Do the actual simulation/analysis.

__module__ = 'compapp.plugins.tests.test_datastores'

compapp.plugins.tests.test_datastores.test_nodir()

compapp.plugins.tests.test_datastores.test_root_exists(tmpdir)

compapp.plugins.tests.test_datastores.test_sub_exists(tmpdir)

compapp.plugins.tests.test_dumpresults module

compapp.plugins.tests.test_logger module

compapp.plugins.tests.test_logger.test_handlers_are_copied()

compapp.plugins.tests.test_logger.test_formatters_are_copied()

```
class compapp.plugins.tests.test_logger.Greeter(*args, **kws)
    Bases: compapp.apps.Computer

    message = 'hello'

    run()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.plugins.tests.test_logger'

class compapp.plugins.tests.test_logger.MyApp(*args, **kws)
    Bases: compapp.plugins.tests.test_logger.Greeter

    class sub(*args, **kws)
        Bases: compapp.plugins.tests.test_logger.Greeter

        class sub(*args, **kws)
            Bases: compapp.plugins.tests.test_logger.Greeter

            __module__ = 'compapp.plugins.tests.test_logger'

            __module__ = 'compapp.plugins.tests.test_logger'

            __module__ = 'compapp.plugins.tests.test_logger'

compapp.plugins.tests.test_logger.assert_levels(output, lvl)
compapp.plugins.tests.test_logger.test_nested_simple_run(capsys)
compapp.plugins.tests.test_logger.test_nested_root_config(capsys)
compapp.plugins.tests.test_logger.test_nested_specific_handler(capsys)
compapp.plugins.tests.test_logger.test_datastore_log(tmpdir)
```

Module contents

Submodules

compapp.plugins.datastores module

```
class compapp.plugins.datastores.BaseDataStore(*args, **kws)
    Bases: compapp.interface.Plugin

    __module__ = 'compapp.plugins.datastores'

compapp.plugins.datastores.iswritable(directory)
    Check if directory is writable.
```

Examples

```
>>> iswritable('.')
True
```

```
>>> os.path.exists('spam')
False
>>> iswritable('spam/egg')
True
```



```
>>> os.access('/', os.W_OK | os.X_OK)
False
>>> os.path.exists('/spam')
False
>>> iswritable('/spam/egg')
False
```

class compapp.plugins.datastores.**DirectoryDataStore**(*args, **kws)

Bases: *compapp.plugins.datastores.BaseDataStore*

Data-store using a directory.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     datastore = DirectoryDataStore
...
>>> mp = MyParametric()
>>> mp.datastore.dir = 'out'
>>> mp.datastore.path('file')
'out/file'
```

path creates intermediate directories if required:

```
>>> os.listdir('.')
['out']
```

If a *nested class* uses *DirectoryDataStore*, the path is automatically allocated under the *dir* of the *owner class*.

```
>>> class MyParametric(Parametric):
...     datastore = DirectoryDataStore
...
...     class nested(Parametric):
...         datastore = DirectoryDataStore
...
>>> mp = MyParametric()
>>> mp.datastore.dir = 'out'
>>> mp.nested.datastore.path()
'out/nested'
>>> mp.nested.datastore.path('file')
'out/nested/file'
>>> mp.nested.datastore.path('dir', 'file')
'out/nested/dir/file'
```

```
>>> mp.nested.datastore.dir = 'another'
>>> mp.nested.datastore.path('file')
'another/file'
```

overwrite = True

clear_before_run = True

on = True

dir

Path to datastore directory (optional).

is_writable()

Return true if *dir* is writable and the owner is in the “run” mode.

It is checked whether the *owner* is in the “run” mode or in the “load” mode here, so that other plugins can avoid overwriting the datastore in the “load” mode.

is_loadable

True if *dir* exists and meta information is stored in there.

prepare()

[to be extended] For a task immediately *after* Executable.prepare.

path(*args, **kws)

Path relative to the base directory *dir*.

Parameters *args* (*str*) – Path relative to *dir*. It will be joined by `os.path.join`.

Keyword Arguments *mkdir* (*bool*) – If `True` (default), make the parent directory of returned *path* (i.e., `os.path.dirname(path)`), not the *path* itself).

Returns *path* – `os.path.join(self.dir, *args)`

Return type *str*

exists(*path)

globitems(pattern)

__module__ = 'compapp.plugins.datastores'

class compapp.plugins.datastores.SubDataStore(*args, **kws)

Bases: `compapp.plugins.datastores.DirectoryDataStore`

Data-store using sub-paths of parent data-store.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     datastore = DirectoryDataStore
...
...     class nested(Parametric):
...         datastore = SubDataStore
...
>>> mp = MyParametric()
>>> mp.datastore.dir = 'out'
>>> mp.nested.datastore.path()
'out/nested'
>>> mp.nested.datastore.path('file')
'out/nested-file'
>>> mp.nested.datastore.path('dir', 'file')
'out/nested-dir/file'
>>> mp.nested.datastore.path('a', 'b', 'c')
'out/nested-a/b/c'
>>> mp.nested.datastore.sep = '.'
>>> mp.nested.datastore.path('file')
'out/nested.file'
```

Since *DirectoryDataStore* already can be used for datastore using sub-directories, this class is specialized for the case using files under the directory of parent datastore. If the *owner class* of this datastore uses only a few files, it makes sense to not allocate a directory and to use this type of datastore.

dir

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             10 = Link('')
...             11 = Link('x')
...             12 = Link('..x')
...             13 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should `path` use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

sep = '-'

path (*args, **kws)

Path relative to the base directory `dir`.

Parameters `args` (*str*) – Path relative to `dir`. It will be joined by `os.path.join`.

Keyword Arguments `mkdir` (*bool*) – If `True` (default), make the parent directory of returned `path` (i.e., `os.path.dirname(path)`), not the `path` itself).

Returns `path` – `os.path.join(self.dir, *args)`

Return type *str*

globitems (*pattern*)

__module__ = 'compapp.plugins.datastores'

`compapp.plugins.datastores.hexdigest` (*jsonable*)

Calculate hex digest of a jsonable object.

```
>>> hexdigest({'a': 1, 'b': 2, 'c': 3})
'e20096b15530bd66a35a7332619f6666e2322070'
```

class `compapp.plugins.datastores.HashDataStore` (*args, **kws)

Bases: `compapp.plugins.datastores.DirectoryDataStore`

Automatically allocated data-store based on hash of parameter.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     datastore = HashDataStore
...     a = 1
...
>>> mp = MyParametric()
>>> mp.datastore.prepare()
>>> mp.datastore.dir
'Data/memo/be/a393597a3c5518cad18a4c96c08d038de3f00a'
>>> mp.a = 2
>>> mp.datastore.prepare()
>>> mp.datastore.dir
'Data/memo/a2/722afcdc53688843b61b8d71329fabab16b6ae'
>>> mp.datastore.basedir = '.'
>>> mp.datastore.prepare()
>>> mp.datastore.dir
'./a2/722afcdc53688843b61b8d71329fabab16b6ae'
```

basedir = 'Data/memo'

__module__ = 'compapp.plugins.datastores'

ownerhash ()

prepare()

[to be extended] For a task immediately *after* Executable.prepare.

compapp.plugins.metastore module

class compapp.plugins.metastore.**MetaStore**(*args, **kws)

Bases: *compapp.interface.Plugin*

metafile = 'meta.json'

datastore

Delegate parameter to its owner.

`x = Delegate()` is equivalent to `x = Link('..x')`.

Examples

```
>>> from compapp.core import Parametric
>>> class Parent(Parametric):
...     class nest1(Parametric):
...         sampling_period = Delegate()
...     class nest2(Parametric):
...         sampling_period = Delegate()
...     sampling_period = 10.0
>>> par = Parent()
>>> par.sampling_period
10.0
>>> par.nest1.sampling_period
10.0
>>> par.nest2.sampling_period
10.0
>>> par.sampling_period = 20.0
>>> par.nest1.sampling_period
20.0
```

log

Delegate parameter to its owner.

`x = Delegate()` is equivalent to `x = Link('..x')`.

Examples

```
>>> from compapp.core import Parametric
>>> class Parent(Parametric):
...     class nest1(Parametric):
...         sampling_period = Delegate()
...     class nest2(Parametric):
...         sampling_period = Delegate()
```

(continues on next page)

(continued from previous page)

```

...
...     sampling_period = 10.0
...
>>> par = Parent()
>>> par.sampling_period
10.0
>>> par.nest1.sampling_period
10.0
>>> par.nest2.sampling_period
10.0
>>> par.sampling_period = 20.0
>>> par.nest1.sampling_period
20.0

```

metafilepath**prepare()**

[to be extended] For a task immediately *after* Executable.prepare.

record(name, data)**load()**

[to be extended] For a task immediately *before* Executable.load.

`__module__ = 'compapp.plugins.metastore'`

compapp.plugins.misc module

class compapp.plugins.misc.Logger(*args, **kws)

Bases: `compapp.interface.Plugin`

Interface to pre-configured `logging.Logger`.

It does the following automatically:

- make a logger with an appropriate dotted name
- set up handler

```

>>> from compapp import Computer
>>> class MyAppForLoggerDemo(Computer):
...     def run(self):
...         self.log.error('error message')
...         self.log.info('info message')
>>> app = MyAppForLoggerDemo()
>>> app.log.formatters['default']['format'] = \
...     '%(levelname)s %(name)s | %(message)s'
>>> app.log.handlers['console']['stream'] = 'ext://sys.stdout'
>>> app.execute()
ERROR compapp.plugins.misc.MyAppForLoggerDemo.0 | error message

```

log

An instance of `logging.Logger`. This is accessible in and after `Executable.run` or `Executable.load` hooks.

Todo: Make it accessible all the time.

critical
error
warn
info
debug

Shortcut for `log.debug` etc.

configurator

Use one of the specified traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     attr = Or(Choice(*'abc'), OfType(int))
...
>>> mp = MyParametric()
>>> mp.attr = 'a'
>>> mp.attr = 1
>>> mp.attr = 1.0
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 1.0
```

Or can be combined with *Link*-like descriptors:

```
>>> from compapp.descriptors import Delegate
>>> class MyRoot(Parametric):
...     attr = Choice(*'abc', nodefault=True)
...
...     class nested(Parametric):
...         attr = Or(OfType(int), Delegate())
...
>>> par = MyRoot()
>>> par.nested.attr
Traceback (most recent call last):
...
AttributeError: 'nested' object has no attribute 'attr'
```

If `par.nested.attr` is not specified (i.e., `OfType(int)` is not in action), accessing to `par.nested.attr` invokes *Delegate* which tries to access `par.attr`. In the above example, `par.attr` was not set so the `AttributeError` was raised. Setting `par.attr` makes `par.nested.attr` accessible:

```
>>> par.attr = 'b'
>>> par.nested.attr
'b'
```

Specifying `par.nested.attr` directly invokes *OfType*:

```
>>> par.nested.attr = 'c'
Traceback (most recent call last):
...
ValueError: None of the traits are matched to: 'c'
>>> par.nested.attr = 496
```

(continues on next page)

(continued from previous page)

```
>>> par.nested.attr
496
```

handlers

The dictionary *handlers* of the [logging configuration dictionary](#). Each handler may contain a special configuration `'datastore': True` indicating that this handler is configured only if datastore is accessible. For nested classes, this attribute is ignored if *ownconfig* is not `True` (default is `'auto'`).

formatters

The dictionary *formatters* of the [logging configuration dictionary](#). See also *handlers*.

ownconfig

If `'auto'` (default), configure handlers if the owner is the root app and reuse the handlers of owner's logger otherwise. The value `True` forces this logger plugin to make own handlers. The value `False` forces the reuse (which is an error if the owner is the root app).

level

Logger level. Default is `'ERROR'`.

critical

“Link” parameter.

It's like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
```

(continues on next page)

(continued from previous page)

```

>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

error

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1

```

(continues on next page)

(continued from previous page)

```

>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

warn

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('..nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('..nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1

```

(continues on next page)

(continued from previous page)

```

>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

info

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True

```

(continues on next page)

(continued from previous page)

```
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

debug

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
```

(continues on next page)

(continued from previous page)

```

>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

datastore

Delegate parameter to its owner.

`x = Delegate()` is equivalent to `x = Link('..x')`.

Examples

```

>>> from compapp.core import Parametric
>>> class Parent(Parametric):
...
...     class nest1(Parametric):
...         sampling_period = Delegate()
...
...     class nest2(Parametric):
...         sampling_period = Delegate()
...
...     sampling_period = 10.0
...
>>> par = Parent()
>>> par.sampling_period
10.0
>>> par.nest1.sampling_period
10.0
>>> par.nest2.sampling_period
10.0
>>> par.sampling_period = 20.0
>>> par.nest1.sampling_period
20.0

```

```
nametemplate = '{self.__class__.__module__}.{self.__class__.__name__}.{id}'
prepare()
    [to be extended] For a task immediately after Executable.prepare.
configure()
should_configure()
__module__ = 'compapp.plugins.misc'
class compapp.plugins.misc.DebugNS(debug)
    Bases: object
    __init__(debug)
        x.__init__(...) initializes x; see help(type(x)) for signature
    __setattr__(name, value)
        x.__setattr__('name', value) <==> x.name = value
    __dict__ = dict_proxy({'__module__': 'compapp.plugins.misc', '__setattr__': <function
    __module__ = 'compapp.plugins.misc'
    __weakref__
        list of weak references to the object (if defined)
```

```
class compapp.plugins.misc.Debug(*args, **kws)
    Bases: compapp.interface.Plugin
```

Debug helper plugin.

Since *Computer* does not allow random attributes to be set, it's hard to debug or interactively investigate simulation and analysis by storing temporary variable to the *Computer* instance. *Debug* provides the place for that.

- If store flag is *not* set, it does not store anything; assignment would be just no-op. This is useful for debugging memory-consuming object.
- If logger is defined, and its level is DEBUG, assignment to *Debug* object writes out debug message.

Example

```
>>> from compapp import Computer
>>> class MySimulator(Computer):
...
...     def run(self):
...         tmp = list(range(3))
...         self.dbg.tmp = tmp
...         self.results.div = [x / len(tmp) for x in tmp]
...
>>> app = MySimulator()
>>> app.log.level = 'DEBUG'
```

Before `app.execute()`, let's tweak logger output so that output is concise:

```
>>> app.log.formatters['default']['format'] = \
...     '%(levelname)s %(name)s | %(message)s'
>>> app.log.handlers['console']['stream'] = 'ext://sys.stdout'
```

Finally...:

```
>>> app.execute()
DEBUG compapp.plugins.misc.MySimulator.0 | ...
DEBUG compapp.plugins.misc.MySimulator.0 | tmp = [0, 1, 2]
DEBUG compapp.plugins.misc.MySimulator.0 | ...
```

```
>>> app.dbg.tmp
[0, 1, 2]
```

Note that executing the same app without `.log.level = 'DEBUG'` suppress the logging and storing the temporary variable. The Debug plugin also can be disabled independent of the log level by setting its *enable* to `False`.

```
>>> app = MySimulator()
>>> app.execute()
>>> app.dbg.tmp
Traceback (most recent call last):
...
AttributeError: 'DebugNS' object has no attribute 'tmp'
```

log

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
```

(continues on next page)

(continued from previous page)

```

3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

enable

If 'auto' (default), enable this plugin if the log level of *Logger* plugin is 'DEBUG' or lower. *True* (*False*) enables (disables) this plugin independent of the log level.

`__init__(*args, **kws)`

`x.__init__(...)` initializes x; see `help(type(x))` for signature

`is_logging_debug()`

`is_enabled()`

`__module__ = 'compapp.plugins.misc'`

class `compapp.plugins.misc.Figure(*args, **kws)`

Bases: `compapp.interface.Plugin`

A wrapper around `matplotlib.pyplot.figure`.

Automatically save and (optionally) show the figure at the end of execution.

Examples

Normally, *Figure* is prepared by an *Executable* subclass such as *Computer*. To manually play with *Figure*, it is equivalent to do:

```

>>> figure = Figure()
>>> figure.prepare()

```

Calling a *Figure* instance gives you a matplotlib figure.

```

>>> figure()
<Figure size ... with 0 Axes>

```

It also has *subplots* which is just a thin wrapper around `matplotlib.pyplot.subplots`:

```

>>> fig, (ax1, ax2) = figure.subplots(2)

```


Suppose you have a nested *Computers*:

```
>>> from compapp.apps import Computer
>>> class MyApp(Computer):
...     class sub(Computer):
...         class sub(Computer):
...             pass
```

Global *Figure* plugin configuration The attribute *autoclose* is *Linked* to the plugin of the owner Computer. It makes possible to configure *Figure* plugins “globally”:

```
>>> app = MyApp()
>>> app.sub.figure.autoclose # default value
True
>>> app.figure.autoclose = False
>>> app.sub.figure.autoclose
False
>>> app.sub.sub.figure.autoclose
False
```

A “sub-tree” of the *Computers* can be configured separately:

```
>>> app.sub.figure.autoclose = True
>>> app.sub.sub.figure.autoclose
True
>>> app.figure.autoclose
False
```

Similar global/sub-tree configuration mechanism works for the *show* attribute:

```
>>> app.figure.show # default value
False
>>> app.figure.show = True
>>> app.sub.figure.show
True
>>> app.sub.sub.figure.show
True
```

```
>>> import os
>>> from compapp import Computer
>>> class MyApp(Computer):
...     def run(self):
...         self.figure()
...         self.figure(name='alpha')
...         self.figure(name='beta')
...
>>> app = MyApp()
>>> app.datastore.dir = 'out'
>>> app.execute()
>>> sorted(f for f in os.listdir('out') if f.startswith('figure-'))
['figure-0.png', 'figure-alpha.png', 'figure-beta.png']
```

See also:

Example: compapp.samples.simple_plots, Example: compapp.samples.named_figure

ext = 'png'

datastore

Delegated attribute accessing to a *BaseDataStore*.

show

Automatically call `matplotlib.pyplot.show` if `True`. The default is `False`, if not specified by the *owner class*. See: *Global Figure plugin configuration*.

autoclose

Automatically close matplotlib figures if `True` which is the default, if not specified by the *owner class*. See: *Global Figure plugin configuration*.

prepare()

[to be extended] For a task immediately *after* `Executable.prepare`.

register (*fig*, *name=None*)

Register *fig* so that it is automatically saved, showed and closed.

__call__ (**args*, ***kws*)

Make a new matplotlib figure.

All the positional and keyword arguments are passed to `matplotlib.pyplot.figure` except one keyword argument *name*.

Parameters *name* (*str*, *int*, or *None*) – Unique name of the figure. It is an error to call this method with the same *name* twice.

Returns *fig*

Return type `matplotlib.figure.Figure`

__getitem__ (*name*)

Make a new matplotlib figure or get the new one.

Examples

The usual preparation (see: *Figure*):

```
>>> figure = Figure()
>>> figure.prepare()
```

If figure is created with a name, it can be accessed by the dict-like interface:

```
>>> fig_a = figure(name='a')
>>> assert fig_a is figure['a'] # access an existing figure
>>> fig_b = figure['b'] # create a new figure
>>> assert fig_b is figure['b']
```

Note that a new figure is created always when *name* is `None`:

```
>>> assert figure[None] is not figure[None]
```

Parameters *name* (*str*, *int*, or *None*) – Unique name of the figure.

Returns *fig*

Return type `matplotlib.figure.Figure`

save()

[to be extended] For a task immediately *after* `Executable.save`.

finish()

[to be extended] For a task immediately *before* Executable.finish.

subplots(*args, **kws)

Wrapper around `matplotlib.pyplot.subplot`.

All the positional and keyword arguments are passed to `matplotlib.pyplot.subplot` except one keyword argument *name*.

Parameters *name* (*str*, *int*, or *None*) – Unique name of the figure. It is an error to call this method with the same *name* twice.

Returns

- *fig* (`matplotlib.figure.Figure`)
- *ax* (`matplotlib.axes.Axes` or *tuple*)

`__module__ = 'compapp.plugins.misc'`

`compapp.plugins.misc.is_runnable(excbl)`

class `compapp.plugins.misc.AutoUpstreams(*args, **kws)`

Bases: `compapp.interface.Plugin`

Automatically execute upstreams.

`__module__ = 'compapp.plugins.misc'`

static `is_runnable(excbl)`

prepare()

[to be extended] For a task immediately *after* Executable.prepare.

class `compapp.plugins.misc.PluginWrapper(*args, **kws)`

Bases: `compapp.interface.Plugin`

Combine multiple plugins into one plugin.

Some plugin such as `AutoDump` has no “interface” and just works behind-the-scene. In this case, having `MyApp.autodump = AutoDump` is just wasting user’s name-space. This class avoid this by moving all those behind-the-scene plugins into one name. See `Computer.plugin` for its use-case.

`__module__ = 'compapp.plugins.misc'`

finish()

load()

post_run()

pre_run()

prepare()

save()

`__getattr__(name)`

`compapp.plugins.misc.makemethod(method)`

`compapp.plugins.misc.real_owner(self)`

compapp.plugins.programinfo module

class compapp.plugins.programinfo.**RecordProgramInfo**(*args, **kwds)

Bases: *compapp.interface.Plugin*

meta

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should `path` use more explicit notation as in the JSON pointer? That is to say, use `'#'` instead of `' '` (an empty string) to indicate the root. Then, for example, `'x'` would be written as `'#.x'`.

```
pre_run()
    [to be extended] For a task immediately before Executable.run.

__module__ = 'compapp.plugins.programinfo'
```

compapp.plugins.recorders module

class `compapp.plugins.recorders.DumpResults(*args, **kws)`

Bases: `compapp.interface.Plugin`

Automatically save owner's results.

Supported back-ends:

- `json` for `dict`, `list` or `tuple`
- `numpy.savez` for `numpy.ndarray`
- `pandas.HDFStore` for `pandas` object

Example

```
>>> import numpy
>>> import pandas
>>> from compapp.apps import Computer
```

Examples below use an app class derived from `Computer` since it is bundled with the `DumpResults` plugin:

```
>>> Computer.magics.dumpresults
<class 'compapp.plugins.recorders.DumpResults'>
```

```
>>> class MyApp(Computer):
...     def run(self):
...         self.results.int = 0
...         self.results.float = 1.0
...         self.results.list = [2, 3, 4]
...         self.results.tuple = (5, 6, 7)
...         self.results.array = numpy.arange(8)
...         self.results.df = pandas.DataFrame({'col': [9, 10, 11]})
...         self.isrun = True
...     isrun = False
...
>>> app = MyApp()
>>> app.datastore.dir = 'out'
>>> os.path.exists(app.datastore.dir)
False
>>> app.execute()
>>> app.isrun
True
>>> os.path.exists(app.datastore.dir)
True
```

The results of the above “simulation” is saved in `results.*` under `app.datastore.dir`:

```
>>> from glob import glob
>>> sorted(glob(app.datastore.path('results.*'))
['out/results.hdf5', 'out/results.json', 'out/results.npz']
```

Now let’s load these results.

```
>>> app2 = MyApp()
>>> app2.mode = 'load'
>>> app2.datastore.dir = 'out'
>>> app2.execute()
>>> app2.isrun
False
>>> sorted(app2.results)
['array', 'df', 'float', 'int', 'list', 'tuple']
>>> app2.results.list
[2, 3, 4]
>>> app2.results.array
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> app2.results.df
   col
0     9
1    10
2    11
```

Note that simple Python types are saved as JSON. Thus, some type information is stripped off after reloading:

```
>>> app2.results.tuple
[5, 6, 7]
```

result_names = ('results',)

These attributes of the owner class are dumped.

pre_run()

[to be extended] For a task immediately before `Executable.run`.

save()

[to be extended] For a task immediately *after* `Executable.save`.

save_now()

Save results to disk immediately.

classmethod save_results (*owner, name*)

static save_results_json (*data, path*)

static save_results_npz (*data, path*)

static save_results_hdf5 (*data, path*)

Save pandas objects in *data* to a HDFStore at *path*.

Implementation notes

It is important to note that repeatedly overwrite the same HDF file is a bad practice, since deleting objects in HDF does **not** free the space:

Can you delete objects in an HDF5 file ? If yes, how ?

Yes, you can use the `H5Ldelete` function to delete objects in an HDF5 file (for HDF5 1.6, use `H5Gunlink`). Currently, however, **the space where the object was located in the file does not**

get re-used. Therefore the size of the file will remain the same. You can get rid of this unused space in a file by writing the contents of the HDF5 file to a new file.

—<http://ns1.hdfgroup.org/hdf5-quest.html#del>

found via [HDFStore file size mysteriously increases - Issue #2132 - pandas-dev/pandas](#)

That's why this function first creates a new HDF file at `path + '.new'` and then move it to `path`.

```
load()
    [to be extended] For a task immediately before Executable.load.
```

```
static load_results_json(path)
```

```
static load_results_npz(path)
```

```
static load_results_hdf5(path)
```

```
save_maybe()
```

Save results in a background thread, if the thread is not busy.

```
__module__ = 'compapp.plugins.recorders'
```

```
class compapp.plugins.recorders.DumpParameters(*args, **kws)
```

Bases: [compapp.interface.Plugin](#)

Dump parameters used for its owner.

Example

```
>>> from compapp.core import Parametric
>>> from compapp.apps import Computer
```

Examples below use an app class derived from [Computer](#) since it is bundled with the [DumpResults](#) plugin:

```
>>> Computer.magics.dumpparameters
<class 'compapp.plugins.recorders.DumpParameters'>
```

```
>>> class MyApp(Computer):
...     i = 0
...     x = 1.0
...     class nested(Parametric):
...         i = 10
...         x = 11.0
...
>>> app = MyApp()
>>> app.datastore.dir = 'out'
>>> app.i = -1
>>> app.execute()
```

The parameters are dumped automatically to `out/params.json`:

```
>>> app.datastore.path('params.json')
'out/params.json'
>>> os.path.exists('out/params.json')
True
```

Running the app using `mode='load'` automatically loads the parameters:

```
>>> app2 = MyApp()
>>> app2.mode = 'load'
>>> app2.datastore.dir = 'out'
>>> app2.execute()
>>> app2.i
-1
```

pre_run()

[to be extended] For a task immediately before Executable.run.

__module__ = 'compapp.plugins.recorders'

load()

[to be extended] For a task immediately *before* Executable.load.

compapp.plugins.sysinfo module

class compapp.plugins.sysinfo.**RecordSysInfo**(*args, **kws)

Bases: *compapp.interface.Plugin*

meta

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         10 = Link('')
...         11 = Link('x')
...         12 = Link('..x')
...         13 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             10 = Link('')
...             11 = Link('x')
...             12 = Link('..x')
...             13 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
```

(continues on next page)

(continued from previous page)

```

>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

root

An alias of `Link('')`.

pre_run()

[to be extended] For a task immediately before `Executable.run`.

`__module__ = 'compapp.plugins.sysinfo'`

compapp.plugins.timing module

`compapp.plugins.timing.getusage_self()`

See: `getusage(2)`

`compapp.plugins.timing.gettimings()`

class `compapp.plugins.timing.RecordTiming(*args, **kws)`

Bases: `compapp.interface.Plugin`

Record timing information.

```

>>> import time
>>> from compapp.apps import Computer
>>> class MyApp(Computer):
...     def run(self):
...         time.sleep(0.5)
>>> app = MyApp()
>>> app.execute()

>>> timing = app.magics.meta.data['timing']
>>> (timing['post']['rusage']['ru_utime']
... + timing['post']['rusage']['ru_stime']
... - timing['pre']['rusage']['ru_utime']
... - timing['pre']['rusage']['ru_stime']) < 1.5
True

```

(continues on next page)

(continued from previous page)

```
>>> 0.5 < timing['post']['time'] - timing['pre']['time'] < 1.5
True
```

meta

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should `path` use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```
pre_run()
    [to be extended] For a task immediately before Executable.run.

__module__ = 'compapp.plugins.timing'

post_run()
    [to be extended] For a task immediately after Executable.run.
```

compapp.plugins.vcs module

```
class compapp.plugins.vcs.Git(filepath)
    Bases: object

    vcstype = 'git'
    git_cmd = ('git',)

    __init__(filepath)
        x.__init__(...) initializes x; see help(type(x)) for signature

    git(*cmd)

    revision()

    isclean()

    vcsinfo()

    __dict__ = dict_proxy({'__module__': 'compapp.plugins.vcs', 'git': <function git>, '
    __module__ = 'compapp.plugins.vcs'

    __weakref__
        list of weak references to the object (if defined)

compapp.plugins.vcs.getvcs(filepath)

class compapp.plugins.vcs.RecordVCS(*args, **kws)
    Bases: compapp.interface.Plugin

    Record VCS revision automatically.
```

Example

```
>>> from compapp.apps import Computer
>>> app = Computer()
>>> app.datastore.dir = 'out'
>>> app.execute()
>>> vcsinfo = app.magics.meta.data['vcs']
>>> sorted(vcsinfo)
['filepath', 'isclean', 'revision', 'root', 'vcs']
>>> from compapp import apps
>>> apps.__file__ == vcsinfo['filepath']
True
```

(continues on next page)

(continued from previous page)

```
>>> vcsinfo['vcs']
'git'
```

VCS information is loaded when the app is executed in load mode:

```
>>> app2 = Computer()
>>> app2.mode = 'load'
>>> app2.datastore.dir = 'out'
>>> app2.execute()
>>> app2.magics.meta.data['vcs'] == vcsinfo
True
```

meta

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```

pre_run()
    [to be extended] For a task immediately before Executable.run.

__module__ = 'compapp.plugins.vcs'

```

Module contents

8.1.3 compapp.samples package

Submodules

compapp.samples.named_figure module

```

class compapp.samples.named_figure.MyApp(*args, **kws)
    Bases: compapp.apps.Computer

    run()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.samples.named_figure'

```

compapp.samples.pluggable_plotter_exec module

```

class compapp.samples.pluggable_plotter_exec.HistPlotter(*args, **kws)
    Bases: compapp.executables.Plotter

    bins = 100

    run(x)
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.samples.pluggable_plotter_exec'

class compapp.samples.pluggable_plotter_exec.CumHistPlotter(*args, **kws)
    Bases: compapp.samples.pluggable_plotter_exec.HistPlotter

    cumulative = True

    __module__ = 'compapp.samples.pluggable_plotter_exec'

class compapp.samples.pluggable_plotter_exec.MySimulator(*args, **kws)
    Bases: compapp.apps.Computer

```

```
samples = 1000

run()
    [to be extended] Do the actual simulation/analysis.

__module__ = 'compapp.samples.pluggable_plotter_exec'

class compapp.samples.pluggable_plotter_exec.MyApp(*args, **kws)
    Bases: compapp.apps.Computer

    sim
        alias of MySimulator

    density
        alias of HistPlotter

    cumdist
        alias of CumHistPlotter

    run()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.samples.pluggable_plotter_exec'
```

compapp.samples.pluggable_plotter_link module

```
class compapp.samples.pluggable_plotter_link.HistPlotter(*args, **kws)
    Bases: compapp.executables.Plotter

    bins = 100

    data
        “Link” parameter.

        It’s like symbolic-link in the file system.
```

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
```

(continues on next page)

(continued from previous page)

```

...
>>> par = MyParametric()
>>> par is par.nest.10 is par.nest.nest.10
True
>>> par.nest.11
1
>>> par.nest.12
1
>>> par.nest.13
3
>>> par.nest.nest.11
1
>>> par.nest.nest.12
2
>>> par.nest.nest.13
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute '13'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```

run()
    [to be extended] Do the actual simulation/analysis.

__module__ = 'compapp.samples.pluggable_plotter_link'

class compapp.samples.pluggable_plotter_link.CumHistPlotter(*args, **kws)
    Bases: compapp.samples.pluggable_plotter_link.HistPlotter

    cumulative = True

    __module__ = 'compapp.samples.pluggable_plotter_link'

class compapp.samples.pluggable_plotter_link.MySimulator(*args, **kws)
    Bases: compapp.apps.Computer

    samples = 1000

    run()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.samples.pluggable_plotter_link'

class compapp.samples.pluggable_plotter_link.MyApp(*args, **kws)
    Bases: compapp.apps.Computer

    autoupstreams
        alias of compapp.plugins.misc.AutoUpstreams

    sim
        alias of MySimulator

```

```
density
    alias of HistPlotter

cumdist
    alias of CumHistPlotter

__module__ = 'compapp.samples.pluggable_plotter_link'
```

compapp.samples.simple_plots module

```
class compapp.samples.simple_plots.MyApp(*args, **kws)
    Bases: compapp.apps.Computer

    run()
        [to be extended] Do the actual simulation/analysis.

    __module__ = 'compapp.samples.simple_plots'
```

Module contents

8.1.4 compapp.tests package

Submodules

compapp.tests.test_apps module

compapp.tests.test_base module

```
compapp.tests.test_base.test_unspecified_pickleable()
compapp.tests.test_base.test_unspecified_repr()
compapp.tests.test_base.test_unspecified_bool()
```

compapp.tests.test_core module

```
class compapp.tests.test_core.ParWithMissingLink(*args, **kws)
    Bases: compapp.core.Parametric

    sub
        alias of compapp.core.Parametric

    link
        “Link” parameter.

        It’s like symbolic-link in the file system.
```

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
```

(continues on next page)

(continued from previous page)

```

...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('..nest.x')
...         broken = Link('..broken')
...
...     class nest(Parametric):
...         x = 3
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('..nest.x')
...         broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False

```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

```

__module__ = 'compapp.tests.test_core'
compapp.tests.test_core.test_missing_link()

```

compapp.tests.test_executable module

compapp.tests.test_interface module

compapp.tests.test_loader module

`compapp.tests.test_loader.test_load_dynamic_class(tmpdir)`

compapp.tests.test_parametric module

```
class compapp.tests.test_parametric.MyParametric(*args, **kws)
    Bases: compapp.core.Parametric

    i = 1
    x = 2.0

    class nested(*args, **kws)
        Bases: compapp.core.Parametric

        a = 100
        b = 200

        __module__ = 'compapp.tests.test_parametric'
    __module__ = 'compapp.tests.test_parametric'

class compapp.tests.test_parametric.TestMixObject(methodName='runTest')
    Bases: unittest.case.TestCase

    class AutoMixed(*args, **kws)
        Bases: compapp.tests.test_parametric.MyParametric

        class nested
            Bases: object

            a = -1

            __dict__ = dict_proxy({'a': -1, '__dict__': <attribute '__dict__' of 'nested'
            __module__ = 'compapp.tests.test_parametric'
            __weakref__
                list of weak references to the object (if defined)

            __module__ = 'compapp.tests.test_parametric'

        test_default_value()

        test_custom_value()

        __module__ = 'compapp.tests.test_parametric'

class compapp.tests.test_parametric.TestMixClassObj(methodName='runTest')
    Bases: compapp.tests.test_parametric.TestMixObject

    class AutoMixed(*args, **kws)
        Bases: compapp.tests.test_parametric.MyParametric

        class nested
```

```
    a = -1
    __module__ = 'compapp.tests.test_parametric'
    __module__ = 'compapp.tests.test_parametric'
    __module__ = 'compapp.tests.test_parametric'
compapp.tests.test_parametric.test_assembler_init()
```

compapp.tests.test_parser module

compapp.tests.test_samples module

compapp.tests.test_setters module

```
class compapp.tests.test_setters.MyApp(*args, **kws)
    Bases: compapp.apps.Computer
    dynamic
        Placeholder for an instance of the class specified by ClassPath.
        The actual instantiation process is delayed until it is accessed (i.e., __get__ method is called).
    classpath
        Class path descriptor that imports specified class on change.
    __module__ = 'compapp.tests.test_setters'
class compapp.tests.test_setters.ClassA(*args, **kws)
    Bases: compapp.apps.Computer
    x = 0
    __module__ = 'compapp.tests.test_setters'
class compapp.tests.test_setters.ClassB(*args, **kws)
    Bases: compapp.tests.test_setters.ClassA
    x = 2
    __module__ = 'compapp.tests.test_setters'
compapp.tests.test_setters.test_subs_dynamic_class()
```

compapp.tests.test_strict module

compapp.tests.test_variator module

Module contents

8.1.5 compapp.utils package

Submodules

compapp.utils.files module

`compapp.utils.files.safewrite(*args, **kws)`
Open a temporary file and replace it with `path` upon close.

Examples

```
>>> with open('data.txt', 'w') as f:
...     _ = f.write('original content')
>>> with safewrite('data.txt') as f:
...     _ = f.write(str(1 / 0))
Traceback (most recent call last):
...
ZeroDivisionError: ...
>>> with open('data.txt') as f:
...     print(f.read())
original content
```

If it were a normal `open`, then the original content would be wiped out.

```
>>> with open('data.txt', 'w') as f:
...     _ = f.write(str(1 / 0))
Traceback (most recent call last):
...
ZeroDivisionError: ...
>>> with open('data.txt') as f:
...     print(f.read())
```

compapp.utils.importer module

`compapp.utils.importer.import_object(name, current_module=None)`
Import an object at “dotted path”.

```
>>> from importlib import import_module
>>> import_object('importlib.import_module') is import_module
True
```

Any object such as package, module, sub-module, function and class can be imported:

```
>>> import_object('json')
<module 'json' from '...'>
```

(continues on next page)

(continued from previous page)

```
>>> import_object('json.load')
<function load at ...>
>>> import_object('os.path')
<module ...>
>>> import_object('os.path.join')
<function ...join at ...>
```

To support relative import, `current_module` has to be provided. Typically, it's `__name__`:

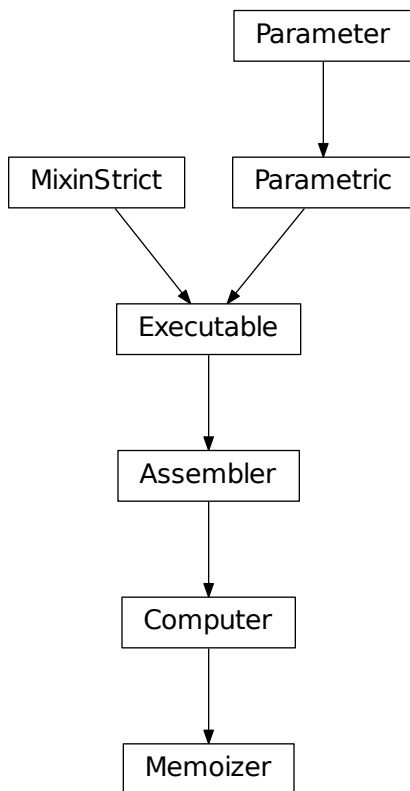
```
import_object('spam.egg', __name__)
```

Module contents

8.2 Submodules

8.2.1 compapp.apps module

Application base classes.



`compapp.apps.print_full_help(app)`

```
class compapp.apps.Computer (*args, **kws)
```

Bases: `compapp.executables.Assembler`

Application base class.

Everything needed for save/load functions are implemented by plugins (see: *magics*). Only the method `run()` is needed to be implemented most of the cases.

```
cli (args=None)
```

Run Command Line Interface of this class.

Examples

```
myapp.py --datastore.dir OUT
myapp.py --sub.floats[0] 10
myapp.py --sub.plotters[0].bin 100
myapp.py --params parameters.yaml
```

If multiple parameter files are given, they will be mixed together before given to the class:

```
myapp.py --params base.yaml extension.yaml
```

Data file for nested classes (:file= modifier):

```
myapp.py --simulator:file=param.yaml --plotter:file=plotter.json
```

Literal eval (:leval= modifier):

```
myapp.py --alpha:leval='2**3'
```

Regular help:

```
myapp.py -h
myapp.py --help
```

Full Help (include full list of parameters):

```
myapp.py -H
myapp.py --help-all
```

```
classmethod get_parser()
```

```
classmethod from_param(param, require_all=False, filter_param=False)
```

Create an instance based on param and execute it.

Parameters

- **param** (*dict*) –
- **require_all** (*bool*) – If true, a `ValueError` is raised if param does not have all parameters for this and nested `Parametric` classes.
- **filter_param** (*bool*) – If true, keys and sub-keys in param which is not valid to this class are removed.

Returns `self`

Return type `Computer`

```
__module__ = 'compapp.apps'
```

```
class compapp.apps.Memoizer(*args, **kws)
    Bases: compapp.apps.Computer
    Computer with HashDataStore.
```

Examples

```
>>> class UpStream1(Memoizer):
...     parameter = 1
...     isrun = None
...
...     def run(self):
...         self.results.data = 'result of intense computation'
...         self.results.name = type(self).__name__
...         self.isrun = True
...
...     def load(self):
...         self.isrun = False
...
>>> class UpStream2(UpStream1):
...     pass
...
>>> class DownStream(UpStream1):
...     up1 = UpStream1
...     up2 = UpStream2
...
...     def run(self):
...         self.up1.execute()
...         self.up2.execute()
...         super(DownStream, self).run()
...         self.results.sum = (self.up1.results.data +
...                             self.up2.results.data)
...
>>> up1 = UpStream1()
>>> up1.execute()
>>> assert up1.isrun
>>> up1 = UpStream1()
>>> up1.execute()
>>> assert not up1.isrun
>>> down = DownStream()
>>> down.execute()
>>> assert not down.up1.isrun
>>> assert down.up2.isrun
>>> assert down.isrun
>>> down.up1.results.data == 'result of intense computation'
True
```

mode

Attribute accepting only one of the specified value.

Parameters

- **default** – Default value (see *nodefault*).
- ***choices** – Alternative values.
- **nodefault** (*bool*) – Treat *default* is just an alternative; i.e., do not “initialize” the attribute.

- ****kwds** – See: *DataDescriptor*.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     choice = Choice(1, 2, 'a')
...
>>> mp = MyParametric()
>>> mp.choice
1
>>> mp.choice = 2
>>> mp.choice = 'a'
>>> mp.choice = 'unknown'
Traceback (most recent call last):
...
ValueError: MyParametric.choice only accepts one of (1, 2, 'a'):
got 'unknown'
```

__module__ = 'compapp.apps'

datastore

alias of *compapp.plugins.datastores.HashDataStore*

8.2.2 compapp.base module

compapp.base.constant (*cls*)

Pickleable singleton generator.

Usage:

```
>>> @constant
... class MyConstant(object):
...     pass
...
>>> MyConstant
MyConstant
```

exception compapp.base.MultiException (*message='Multiple exceptions are raised:', errors=None*)

Bases: *exceptions.Exception*

__init__ (*message='Multiple exceptions are raised:', errors=None*)

x.__init__(...) initializes *x*; see *help(type(x))* for signature

__str__ () <==> *str(x)*

record (***kwds*)

classmethod recorder (***kwds*)

Context manager to raise error (if any) at the end of execution.

```
>>> with MultiException.recorder() as mexc:
...     with mexc.record():
...         raise ValueError(1)
...     with mexc.record():
...         raise RuntimeError(2)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
MultiException: Multiple exceptions are raised:
* ValueError: 1
* RuntimeError: 2
```

classmethod `run(callbacks)`

Run all callbacks ignoring exceptions and raise at the end (if any).

```
>>> def raiser(e):
...     raise e
>>> MultiException.run([
...     # takes tuple (func, arg1, arg2, ...):
...     (raiser, ValueError(1)),
...     # or a function without any argument:
...     lambda: raiser(RuntimeError(2)),
... ])
Traceback (most recent call last):
...
MultiException: Multiple exceptions are raised:
* ValueError: 1
* RuntimeError: 2
```

`__module__ = 'compapp.base'``__weakref__`

list of weak references to the object (if defined)

class `compapp.base.DictObject(*args, **kws)`Bases: `object`

Dict-Object Hybrid

Example

```
>>> obj = DictObject()
>>> obj['mykey'] = 'myval'
>>> obj.mykey
'myval'
>>> obj()
{'mykey': 'myval'}
>>> obj
DictObject({'mykey': 'myval'})
>>> 'mykey' in obj
True
>>> '__init__' in obj
False
>>> list(obj)
['mykey']
>>> len(obj)
1
>>> del obj['mykey']
>>> len(obj)
0
>>> obj.mykey = 'anotherval'
>>> obj['mykey']
```

(continues on next page)

(continued from previous page)

```
'anotherval'
>>> del obj.mykey
>>> len(obj)
0
>>> obj = DictObject()
>>> obj
DictObject({})
```

__init__ (*args, **kwargs)

x.__init__(...) initializes x; see help(type(x)) for signature

__getitem__ (name)

__setitem__ (name, value)

__call__ (...) <==> x(...)

__iter__ ()

__delitem__ (x)

__contains__ (x)

__len__ ()

__repr__ () <==> repr(x)

__eq__ (other)

x.__eq__(y) <==> x==y

__dict__ = dict_proxy({'__delitem__': <function __delitem__>, '__module__': 'compapp

__module__ = 'compapp.base'

__weakref__

list of weak references to the object (if defined)

compapp.base.**dotted_to_nested** (ddict)

Convert dotted dictionary to nested dictionary.

```
>>> assert dotted_to_nested({
...     'a.b.c': 1,
...     'a.d': 2,
... }) == dict(
...     a=dict(b=dict(c=1),
...             d=2),
... )
```

compapp.base.**deepmixdicts** (*dicts)

Deeply mix dictionaries.

```
>>> deepmixdicts(
...     {'a': 1, 'b': 2, 'sub': {'x': 1, 'y': 2, 'sub': {'v': 1}}},
...     {'a': 2, 'c': 3, 'sub': {'x': 2, 'z': 3, 'sub': {'v': 2}}},
... ) == {
...     'a': 2, 'b': 2, 'c': 3,
...     'sub': {
...         'x': 2, 'y': 2, 'z': 3,
...         'sub': {'v': 2},
...     },
... }
```

(continues on next page)

(continued from previous page)

```
... }
True
```

`compapp.base.itervars` (*obj*)

`compapp.base.props_of` (*obj*, *_type*)

Yield attributes of type *_type* which already exist at the “class-level”.

`compapp.base.mixdicts` (*dicts*)

8.2.3 compapp.cli module

`compapp.cli.import_appclass` (*path*, *supcls*=<class 'compapp.apps.Computer'>)

`compapp.cli.cli_run` (*path*, *args*)

Run application class at *path* with *args*.

Examples

```
run compapp.samples.pluggable_plotter_exec.MyApp
```

When there is only one `compapp.Computer` subclass in a module, class name can be omitted. Thus, these are equivalent:

```
run compapp.samples.simple_plots.MyApp
run compapp.samples.simple_plots
```

If there are multiple classes, a “main” class for a module can be declared simply by setting it to a variable called `main`, i.e., by `main = MyApp`.

When `A/B/C/D.py` is passed as *path* and `A/B/` (say) is a project root, i.e., there is a `A/B/setup.py` file, then it is equivalent to passing the module path `C.D`. Thus, the following is equivalent to above two:

```
run compapp/samples/simple_plots.py
```

`compapp.cli.cli_mrun` (*path*, *args*)

Run any class at *path* with different parameters.

See help of `run` command for how *path* is interpreted. Note that *path* can point to any class (not just `Computer` subclass).

`compapp.cli.make_parser` (*doc*=None)

`compapp.cli.main` (*args*=None)

8.2.4 compapp.conftest module

8.2.5 compapp.core module

`compapp.core.basic_types` = (<type 'complex'>, <type 'float'>, <type 'int'>, <type 'str'>, ...)

Basic Python types.

`compapp.core.simple_type_check` (*default*, *actual*, *errfmt*, ***fmtkws*)

`compapp.core.automixin` (*owner*, *key*, *params*)

```
class compapp.core.WeakRefProperty(name)
    Bases: object

    __init__(name)
        x.__init__(...) initializes x; see help(type(x)) for signature

    __get__(obj, cls)

    __set__(obj, value)

    __dict__ = dict_proxy({'__module__': 'compapp.core', '__set__': <function __set__>,
    __module__ = 'compapp.core'

    __weakref__
        list of weak references to the object (if defined)

class compapp.core.Private
    Bases: object

    owner

    __init__()
        x.__init__(...) initializes x; see help(type(x)) for signature

    set_context(owner, myname)

    getroot()

    __dict__ = dict_proxy({'__module__': 'compapp.core', 'getroot': <function getroot>,
    __module__ = 'compapp.core'

    __weakref__
        list of weak references to the object (if defined)

class compapp.core.Parameter
    Bases: object

    __dict__ = dict_proxy({'__dict__': <attribute '__dict__' of 'Parameter' objects>, '__
    __module__ = 'compapp.core'

    __weakref__
        list of weak references to the object (if defined)

compapp.core.private(par)
    Access the private data storage for compapp internals.

    Parameters par (Parameter) –

    Returns prv – The Private instance attached to the object par of any subclass of Parameter
        is returned.

    Return type Private

class compapp.core.Descriptor(default=Unspecified, isparam=Unspecified)
    Bases: object

    __init__(default=Unspecified, isparam=Unspecified)
        x.__init__(...) initializes x; see help(type(x)) for signature

    isparam = False

    myname(obj, error=False)

    __get__(obj, cls)
```

```

__dict__ = dict_proxy({'__module__': 'compapp.core', 'myname': <function myname>, 'i
__module__ = 'compapp.core'
__weakref__
    list of weak references to the object (if defined)
class compapp.core.DataDescriptor(**kws)
    Bases: compapp.core.Descriptor
    isparam = True
    __init__(**kws)
        x.__init__(...) initializes x; see help(type(x)) for signature
    get(obj)
    verify(obj, value, myname=None)
    __set__(obj, value)
    __module__ = 'compapp.core'

```

```

class compapp.core.Parametric(*args, **kws)
    Bases: compapp.core.Parameter

```

The basic parametrized class.

Any properties which are a subclass of *Parameter* or whose type are one of *basic_types* are considered as “parameter” to the class.

Note that *Parametric* class itself is a subclass of *Parameter*. Thus, it allows “nested parameter”, i.e., *Parametric* class having *Parametric* property.

Example

```

>>> class MyParametric(Parametric):
...     i = 1
...     x = 2.0
...
...     class param(Parametric):
...         a = 100
...         b = 200
...
>>> mp = MyParametric({'i': 30, 'param': {'a': 0}})
>>> mp.x
2.0
>>> mp.i
30
>>> mp.param.a
0
>>> mp.param.b
200

```

Automatic parameter mix-in. If it is just for resetting parameter, you don’t need to inherit nested *Parametric* classes; it happens automatically.

```

>>> class Base(Parametric):
...     class x(Parametric):
...         i = 0

```

(continues on next page)

(continued from previous page)

```

...         j = 1
...
>>> class Another(Base):
...     class x: # do not need to inherit Base.x
...         i = -1
...
>>> par = Another()
>>> par.x.i
-1
>>> par.x.j
1

```

Automatic type-checking and casting.

```

>>> MyParametric(i='a')
Traceback (most recent call last):
...
ValueError: Value 'a' (type: str) cannot be assigned to the
variable MyParametric.i (default: 1) ...
>>> mp = MyParametric()
>>> mp.x = 1j
Traceback (most recent call last):
...
ValueError: Value 1j (type: complex) cannot be assigned to the
variable MyParametric.x (default: 2.0) ...
>>> MyParametric(x=10).x
10.0

```

__init__ (*args, **kws)
 x.__init__(...) initializes x; see help(type(x)) for signature

__setattr__ (name, value)
 x.__setattr__('name', value) <==> x.name = value

params (nested=False, type=None)
 Get parameters as a dict.

Parameters

- **nested** (*bool*) – if true, return a nested dict for *Parametric* properties.
- **type** – type of parameters to be returned

Returns params

Return type dict

Example

```

>>> class MyParametric(Parametric):
...     x = 1.0
...     i = 1
...     s = 'A'
...
...     class ps(Parametric):
...         y = 2.0
...         j = 2

```

(continues on next page)

(continued from previous page)

```

...
>>> mp = MyParametric()
>>> mp.params() == {'x': 1.0, 'i': 1, 's': 'A'}
True
>>> mp.params(nested=True) == dict(mp.params(), ps=dict(y=2.0, j=2))
True
>>> mp.params(nested=True, type=int) == {'i': 1, 'ps': {'j': 2}}
True

```

classmethod paramnames (*type=None*)

List names of parameter for this class.

User sub-class can modify own behavior by overriding this method.

Example

```

>>> class MyParametric(Parametric):
...     x = 1.0
...     i = 1
...     s = 'string'
...
...     class ps(Parametric):
...         pass
...
>>> sorted(MyParametric.paramnames())
['i', 'ps', 's', 'x']
>>> MyParametric.paramnames(type=float)
['x']

```

classmethod defaultparams (*nested=False, type=None*)

Get default parameters as a `dict`.

Example

```

>>> class MyParametric(Parametric):
...     x = 1.0
...     i = 1
...     s = 'string'
...
...     class ps(Parametric):
...         y = 2.0
...         j = 2
...
>>> shallow = {
...     'x': 1.0, 'i': 1, 's': 'string'
... }
>>> MyParametric.defaultparams() == shallow
True
>>> MyParametric.defaultparams(nested=True) \
...     == dict(shallow, ps={'y': 2.0, 'j': 2})
True
>>> MyParametric.defaultparams(nested=True, type=int) \
...     == {'i': 1, 'ps': {'j': 2}}
True

```

```
__module__ = 'compapp.core'

class compapp.core.Defer
  Bases: object
  Callback registry.
```

Example

```
>>> defer = Defer()
>>> @defer(1, y=2)
... def _(x, y):
...     print('x:', x, ' ', 'y:', y)
...
>>> defer.call()
x: 1   y: 2
```

Once the callbacks are called, they are cleared from the registry.

```
>>> defer.call()
```

Of course, decorator syntax is not mandatory:

```
>>> defer('x:', 1)(print)
<function ...>
>>> defer.call()
x: 1
```

The callback registry can be categorized with keys:

```
>>> for i in range(3):
...     for v in 'xy':
...         _ = defer.keyed(('cat', i), v, '=', i)(print)
...
...
```

One category can be run by specifying key argument of `call`.

```
>>> defer.call(('cat', 1))
y = 1
x = 1
```

Calling without key argument means “call everything”. Note that the category 1 callbacks are not called below, since they were already called.

```
>>> defer.call()
y = 2
x = 2
y = 0
x = 0
```

```
__init__ ()
  x.__init__(...) initializes x; see help(type(x)) for signature

__dict__ = dict_proxy({'__dict__': <attribute '__dict__' of 'Defer' objects>, '__module__': 'compapp.core'})

__module__ = 'compapp.core'
```


__weakref__
list of weak references to the object (if defined)

keyed (*key*, **args*, ***kws*)
Register a callback with *key*.

__call__ (**args*, ***kws*)
Register a callback; equivalent to `.keyed(Unspecified, ...)`

call (*key=None*)
Call deferred callbacks and un-register them.

8.2.6 compapp.executables module

Executable subclasses.

```
class compapp.executables.MagicPlugins (*args, **kws)
    Bases: compapp.plugins.misc.PluginWrapper

    __module__ = 'compapp.executables'

    autoupstreams
        alias of compapp.plugins.misc.AutoUpstreams

    dumpparameters
        alias of compapp.plugins.recorders.DumpParameters

    dumpresults
        alias of compapp.plugins.recorders.DumpResults

    meta
        alias of compapp.plugins.metastore.MetaStore

    programinfo
        alias of compapp.plugins.programinfo.RecordProgramInfo

    recordtiming
        alias of compapp.plugins.timing.RecordTiming

    recordvcs
        alias of compapp.plugins.vcs.RecordVCS

    sysinfo
        alias of compapp.plugins.sysinfo.RecordSysInfo

class compapp.executables.Assembler (*args, **kws)
    Bases: compapp.interface.Executable

    Executable bundled with useful plugins.

    mode
        Execution mode. The mode 'run' and 'load' means to call run and load, respectively. When
        'auto' is specified, call run if is_loadable returns True otherwise call load.

    results
        Attributes set to this property are saved to datastore by DumpResults plugin. Downstream classes
        must rely only on the data under this property. For debugging purpose, use dbg to store intermediate
        variables.

    magics
        Plugins that work behind the scene.

        alias of MagicPlugins
```

dbg

“Link” parameter.

It’s like symbolic-link in the file system.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     x = 1
...     broken = Link('..')
...
...     class nest(Parametric):
...         x = 2
...         l0 = Link('')
...         l1 = Link('x')
...         l2 = Link('..x')
...         l3 = Link('.nest.x')
...         broken = Link('..broken')
...
...         class nest(Parametric):
...             x = 3
...             l0 = Link('')
...             l1 = Link('x')
...             l2 = Link('..x')
...             l3 = Link('.nest.x')
...             broken = Link('..broken')
...
>>> par = MyParametric()
>>> par is par.nest.l0 is par.nest.nest.l0
True
>>> par.nest.l1
1
>>> par.nest.l2
1
>>> par.nest.l3
3
>>> par.nest.nest.l1
1
>>> par.nest.nest.l2
2
>>> par.nest.nest.l3
Traceback (most recent call last):
...
AttributeError: 'nest' object has no attribute 'l3'
>>> hasattr(par, 'broken')
False
>>> hasattr(par.nest, 'broken')
False
>>> hasattr(par.nest.nest, 'broken')
False
```

Todo: Should path use more explicit notation as in the JSON pointer? That is to say, use '#' instead of '' (an empty string) to indicate the root. Then, for example, 'x' would be written as '#.x'.

class figure (*args, **kws)

Bases: *compapp.plugins.misc.Figure*

Figure with *SubDataStore*.

datastore

alias of *compapp.plugins.datastores.SubDataStore*

__module__ = 'compapp.executables'

datastore

alias of *compapp.plugins.datastores.DirectoryDataStore*

should_load()

Return *True* if *load* should be run.

Default is to return *False* always. See also *Assembler.is_loadable*.

is_loadable()

[to be extended] Return *True* if self is loadable.

Default implementation returns the value of *.datastore.is_loadable* which reflects whether or not *params.json* exists in the *.datastore.dir*.

arange

Number of arguments that the *run* method accepts.

```
>>> class MyComp00(Assembler):
...     def run(self):
...         pass
...
>>> MyComp00().arange
(0, 0)
```

```
>>> class MyComp12(Assembler):
...     def run(self, x, y=1):
...         pass
...
>>> MyComp12().arange
(1, 2)
```

```
>>> class MyComp2i(Assembler):
...     def run(self, x, y, *args):
...         pass
...
>>> MyComp2i().arange
(2, None)
```

__module__ = 'compapp.executables'

debug

alias of *compapp.plugins.misc.Debug*

log

alias of *compapp.plugins.misc.Logger*

class compapp.executables.Loader (*args, **kws)

Bases: *compapp.executables.Assembler*

Data source loaded from disk.

```
class magics(*args, **kws)
    Bases: compapp.executables.MagicPlugins

    dumpresults = None

    __module__ = 'compapp.executables'
__module__ = 'compapp.executables'

datastore
    alias of compapp.plugins.datastores.SubDataStore

class compapp.executables.Plotter(*args, **kws)
    Bases: compapp.executables.Assembler

    An Assembler subclass specialized for plotting.
```

Example

```
class DensityPlotter(Plotter):
    pass

class CumulativeDistributionPlotter(Plotter):
    pass

class MyApp(Computer):
    density = DensityPlotter
    dist = CumulativeDistributionPlotter

    def run(self):
        self.simulator.execute()
        self.density.execute(self.simulator)
        self.dist.execute(self.simulator)
```

Then `myapp.py --datastore.dir out/` saves density plot in `out/density-0.png` and cumulative distribution in `out/dist-0.png`.

```
__module__ = 'compapp.executables'

figure
    alias of compapp.plugins.misc.Figure

datastore
    alias of compapp.plugins.datastores.SubDataStore
```

8.2.7 compapp.interactive module

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

This function modify the default for compapp classes (only *Figure*, at the moment) to behave nicely in REPLs.

8.2.8 compapp.interface module

```
class compapp.interface.Executable(*args, **kws)
    Bases: compapp.strict.MixinStrict, compapp.core.Parametric

    The base class supporting execution and plugin mechanism.
```

Example

```
>>> class MyPlugin(Plugin):
...     def pre_run(self):
...         print('Pre-run:', self.__class__.__name__)
...
>>> class MyExec(Executable):
...     myplugin = MyPlugin
...
...     def run(self):
...         print('Run:', self.__class__.__name__)
...
>>> excbl = MyExec()
>>> excbl.execute()
Pre-run: MyPlugin
Run: MyExec
```

__init__ (*args, **kws)
 x.__init__(...) initializes x; see help(type(x)) for signature

defer
 Attribute accepting only certain type(s) of value.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     i = OfType(int)
...
>>> MyParametric(i='a')
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int: got 'a' of type str
```

OfType can take multiple classes:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, float, str)
...
>>> mp = MyParametric()
>>> mp.i = 'a'
>>> mp.i = 1j
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int, float or str: got 1j
of type complex
```

It is an error to access unset an OfType attribute:

```
>>> mp = MyParametric()
>>> mp.i
Traceback (most recent call last):
...
AttributeError: 'MyParametric' object has no attribute 'i'
```

OfType can take *default* argument:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, default=0)
...
>>> mp = MyParametric()
>>> mp.i
0
```

Castable values are cast automatically:

```
>>> class MyParametric(Parametric):
...     x = OfType(float)
...
>>> mp = MyParametric()
>>> mp.x = 1                                # assigning an int...
>>> assert isinstance(mp.x, float)         # ... cast to a float
>>> import numpy
>>> mp.x = numpy.float16(2.0)               # assigning a numpy float...
>>> assert isinstance(mp.x, float)         # ... cast to a float
```

execute (*args)

Execute this instance.

Basically, this function does this:

```
prepare()
Plugin.prepare()
if should_load():
    load()
    Plugin.load()
else:
    Plugin.pre_run()
    run()
    Plugin.post_run()
    save()
    Plugin.save()
Plugin.finish()
finish()
```

Note that `Plugin.<method>()` are run for all plugins.

should_load()

Return `True` if `load` should be run.

Default is to return `False` always. See also `Assembler.is_loadable`.

prepare()

[to be extended] Prepare for `run/load`; e.g., execute upstreams.

run (*args)

[to be extended] Do the actual simulation/analysis.

save()

[to be extended] Save the result manually.

load()

[to be extended] Load saved result manually.

finish()

[to be extended] Do anything to be done before exit.

```
__module__ = 'compapp.interface'
```

```
compapp.interface.call_plugins(self, method)

class compapp.interface.Plugin(*args, **kws)
    Bases: compapp.core.Parametric

    Plugin base class.

    __init__(*args, **kws)
        x.__init__(...) initializes x; see help(type(x)) for signature

    prepare()
        [to be extended] For a task immediately after Executable.prepare.

    pre_run()
        [to be extended] For a task immediately before Executable.run.

    post_run()
        [to be extended] For a task immediately after Executable.run.

    save()
        [to be extended] For a task immediately after Executable.save.

    load()
        [to be extended] For a task immediately before Executable.load.

    __module__ = 'compapp.interface'

    finish()
        [to be extended] For a task immediately before Executable.finish.
```

8.2.9 compapp.loader module

`compapp.loader.load(path)`
 Load result saved at path.
 path can be a path to a datastore directory, a path to a params.json file, or a path to a meta.json file.

Examples

```
>>> from compapp import Computer, load
>>> class MyApp(Computer):
...     x = 1
...     def run(self):
...         self.results.y = self.x ** 2
```

```
>>> app = MyApp()
>>> app.datastore.dir = 'out'
>>> app.execute()
>>> assert load('out').results == app.results
>>> assert load('out/meta.json').results == app.results
>>> assert load('out/params.json').results == app.results
```

8.2.10 compapp.parser module

```
class compapp.parser.Option(lhs, rhs, modifier)
    Bases: tuple
```

```
__dict__ = dict_proxy({'__module__': 'compapp.parser', '__getstate__': <function __g
__getnewargs__ ()
    Return self as a plain tuple. Used by copy and pickle.

__getstate__ ()
    Exclude the OrderedDict from pickling

__module__ = 'compapp.parser'

static __new__ (_cls, lhs, rhs, modifier)
    Create new instance of Option(lhs, rhs, modifier)

__repr__ ()
    Return a nicely formatted representation string

__slots__ = ()

lhs
    Alias for field number 0

modifier
    Alias for field number 2

rhs
    Alias for field number 1

compapp.parser.parse_key (key)

compapp.parser.parse_assignment_options (arguments)
    Parse assignment options (e.g., -dict['key']).

compapp.parser.parse_bool (value)

compapp.parser.parse_value (holder, name, value)

compapp.parser.assign_option (obj, lhs, rhs)

compapp.parser.load_any (path)
    Load any data file from given path.

compapp.parser.process_modifier (lhs, rhs, modifier)

compapp.parser.process_assignment_options (obj, options)

compapp.parser.make_parser (doc=None)

compapp.parser.parseargs (parser, args=None)
```

8.2.11 compapp.setters module

```
compapp.setters.rec_setattr (obj, name, value)
    Recursive version of setattr.
```

If value is a `dict` and the attribute name of `obj` is some non-simple object, set attributes of it by the values of the dictionary value. Otherwise fallbacks to `setattr`.

```
compapp.setters.rec_setattrrs (obj, dct)
    Set values of nested dictionary dct to attributes of obj.
```

```
>>> class A(object):
...     b = c = d = e = f = None
>>> a = A()
```

(continues on next page)

(continued from previous page)

```

>>> a.b = A()
>>> a.b.c = A()
>>> rec_setattr(a, dict(b=dict(c=dict(d=1), e=2), f=3))
>>> a.b.c.d
1
>>> a.b.e
2
>>> a.f
3

```

8.2.12 compapp.strict module

class compapp.strict.MixinStrict

Bases: `object`

`__setattr__` (*name, value*)

`x.__setattr__('name', value) <==> x.name = value`

`__dict__` = `dict_proxy({'__module__': 'compapp.strict', '__setattr__': <function __se`

`__module__` = `'compapp.strict'`

`__weakref__`

list of weak references to the object (if defined)

8.2.13 compapp.testing module

8.2.14 compapp.variator module

class compapp.variator.ParamBuilder (*args, **kws)

Bases: `compapp.core.Parametric`

choices

Attribute accepting only dict with certain traits.

Examples

```

>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
...     strint = Dict(str, int)
...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str
```

ranges

Attribute accepting only dict with certain traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
...     strint = Dict(str, int)
...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str
```

linspace

Attribute accepting only dict with certain traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
...     strint = Dict(str, int)
...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str
```

logspace

Attribute accepting only dict with certain traits.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     anydict = Dict()
...     strint = Dict(str, int)
...
>>> mp = MyParametric()
>>> mp.anydict = {'a': 1}
>>> mp.strint = {'a': 1}
>>> mp.strint = {1: 1}
Traceback (most recent call last):
...
ValueError: MyParametric.strint[...] only accepts type of str: got 1
of type int
>>> mp.strint = {'a': 'b'}
Traceback (most recent call last):
...
ValueError: MyParametric.strint['a'] only accepts type of int: got 'b'
of type str
```

build_params()

keys()

__module__ = 'compapp.variator'

compapp.variator.**execute**(arg)

class compapp.variator.**Variator**(*args, **kws)

Bases: *compapp.apps.Computer*

base

Placeholder for an instance of the class specified by *ClassPath*.

The actual instantiation process is delayed until it is accessed (i.e., `__get__` method is called).

classpath

Class path descriptor that imports specified class on change.

builder

alias of *ParamBuilder*

processes = -1

executor

Attribute accepting only one of the specified value.

Parameters

- **default** – Default value (see *nodefault*).
- ***choices** – Alternative values.
- **nodefault** (*bool*) – Treat *default* is just an alternative; i.e., do not “initialize” the attribute.
- ****kws** – See: *DataDescriptor*.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     choice = Choice(1, 2, 'a')
...
>>> mp = MyParametric()
>>> mp.choice
1
>>> mp.choice = 2
>>> mp.choice = 'a'
>>> mp.choice = 'unknown'
Traceback (most recent call last):
...
ValueError: MyParametric.choice only accepts one of (1, 2, 'a'):
got 'unknown'
```

datastore_format = '{}'

variants

Attribute accepting only certain type(s) of value.

Examples

```
>>> from compapp.core import Parametric
>>> class MyParametric(Parametric):
...     i = OfType(int)
...
>>> MyParametric(i='a')
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int: got 'a' of type str
```

OfType can take multiple classes:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, float, str)
...
>>> mp = MyParametric()
>>> mp.i = 'a'
>>> mp.i = 1j
Traceback (most recent call last):
...
ValueError: MyParametric.i only accepts type of int, float or str: got 1j
of type complex
```

It is an error to access unset an OfType attribute:

```
>>> mp = MyParametric()
>>> mp.i
Traceback (most recent call last):
...
AttributeError: 'MyParametric' object has no attribute 'i'
```

OfType can take *default* argument:

```
>>> class MyParametric(Parametric):
...     i = OfType(int, default=0)
...
>>> mp = MyParametric()
>>> mp.i
0
```

Castable values are cast automatically:

```
>>> class MyParametric(Parametric):
...     x = OfType(float)
...
>>> mp = MyParametric()
>>> mp.x = 1                                # assigning an int...
>>> assert isinstance(mp.x, float)          # ... cast to a float
>>> import numpy
>>> mp.x = numpy.float16(2.0)                # assigning a numpy float...
>>> assert isinstance(mp.x, float)          # ... cast to a float
```

```
run()
    [to be extended] Do the actual simulation/analysis.
__module__ = 'compapp.variator'
```

8.3 Module contents

8.3.1 Bases classes for simulation/analysis “apps”

CHAPTER 9

compapp

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

C

- `compapp`, 113
- `compapp.apps`, 89
- `compapp.base`, 92
- `compapp.cli`, 95
- `compapp.core`, 95
- `compapp.descriptors`, 47
- `compapp.descriptors.dynamic_class`, 36
- `compapp.descriptors.links`, 38
- `compapp.descriptors.misc`, 40
- `compapp.descriptors.tests`, 36
- `compapp.descriptors.tests.test_complex`, 31
- `compapp.descriptors.tests.test_or`, 33
- `compapp.descriptors.tests.test_params`, 34
- `compapp.descriptors.traits`, 41
- `compapp.executables`, 101
- `compapp.interactive`, 104
- `compapp.interface`, 104
- `compapp.loader`, 107
- `compapp.parser`, 107
- `compapp.plugins`, 81
- `compapp.plugins.datastores`, 52
- `compapp.plugins.metastore`, 57
- `compapp.plugins.misc`, 58
- `compapp.plugins.programinfo`, 72
- `compapp.plugins.recorders`, 73
- `compapp.plugins.sysinfo`, 76
- `compapp.plugins.tests`, 52
- `compapp.plugins.tests.test_autoupstreams`, 47
- `compapp.plugins.tests.test_datastores`, 51
- `compapp.plugins.tests.test_logger`, 51
- `compapp.plugins.timing`, 77
- `compapp.plugins.vcs`, 79
- `compapp.samples`, 84
- `compapp.samples.named_figure`, 81
- `compapp.samples.pluggable_plotter_exec`, 81
- `compapp.samples.pluggable_plotter_link`, 82
- `compapp.samples.simple_plots`, 84
- `compapp.setters`, 108
- `compapp.strict`, 109
- `compapp.tests`, 88
- `compapp.tests.test_base`, 84
- `compapp.tests.test_core`, 84
- `compapp.tests.test_loader`, 86
- `compapp.tests.test_parametric`, 86
- `compapp.tests.test_setters`, 87
- `compapp.utils`, 89
- `compapp.utils.files`, 88
- `compapp.utils.importer`, 88
- `compapp.variator`, 109

Symbols

- `__call__()` (compapp.base.DictObject method), 94
- `__call__()` (compapp.core.Defer method), 101
- `__call__()` (compapp.plugins.misc.Figure method), 70
- `__contains__()` (compapp.base.DictObject method), 94
- `__delitem__()` (compapp.base.DictObject method), 94
- `__dict__` (compapp.base.DictObject attribute), 94
- `__dict__` (compapp.core.Defer attribute), 100
- `__dict__` (compapp.core.Descriptor attribute), 96
- `__dict__` (compapp.core.Parameter attribute), 96
- `__dict__` (compapp.core.Private attribute), 96
- `__dict__` (compapp.core.WeakRefProperty attribute), 96
- `__dict__` (compapp.descriptors.tests.test_params.TestDictWithDefault attribute), 35
- `__dict__` (compapp.parser.Option attribute), 107
- `__dict__` (compapp.plugins.misc.DebugNS attribute), 66
- `__dict__` (compapp.plugins.vcs.Git attribute), 79
- `__dict__` (compapp.strict.MixinStrict attribute), 109
- `__dict__` (compapp.tests.test_parametric.TestMixObjectAutomixNested attribute), 86
- `__eq__()` (compapp.base.DictObject method), 94
- `__get__()` (compapp.core.Descriptor method), 96
- `__get__()` (compapp.core.WeakRefProperty method), 96
- `__getattr__()` (compapp.plugins.misc.PluginWrapper method), 71
- `__getitem__()` (compapp.base.DictObject method), 94
- `__getitem__()` (compapp.plugins.misc.Figure method), 70
- `__getnewargs__()` (compapp.parser.Option method), 108
- `__getstate__()` (compapp.parser.Option method), 108
- `__init__()` (compapp.base.DictObject method), 94
- `__init__()` (compapp.base.MultiException method), 92
- `__init__()` (compapp.core.DataDescriptor method), 97
- `__init__()` (compapp.core.Defer method), 100
- `__init__()` (compapp.core.Descriptor method), 96
- `__init__()` (compapp.core.Parametric method), 98
- `__init__()` (compapp.core.Private method), 96
- `__init__()` (compapp.core.WeakRefProperty method), 96
- `__init__()` (compapp.descriptors.dynamic_class.ClassPath method), 36
- `__init__()` (compapp.descriptors.dynamic_class.ClassPlaceholder method), 37
- `__init__()` (compapp.descriptors.links.Delegate method), 40
- `__init__()` (compapp.descriptors.links.Link method), 39
- `__init__()` (compapp.descriptors.links.Root method), 39
- `__init__()` (compapp.descriptors.misc.Constant method), 40
- `__init__()` (compapp.descriptors.traits.Choice method), 45
- `__init__()` (compapp.descriptors.traits.Dict method), 44
- `__init__()` (compapp.descriptors.traits.List method), 43
- `__init__()` (compapp.descriptors.traits.OfType method), 42
- `__init__()` (compapp.descriptors.traits.Or method), 46
- `__init__()` (compapp.descriptors.traits.Required method), 42
- `__init__()` (compapp.interface.Executable method), 105
- `__init__()` (compapp.interface.Plugin method), 107
- `__init__()` (compapp.plugins.misc.Debug method), 68
- `__init__()` (compapp.plugins.misc.DebugNS method), 66
- `__init__()` (compapp.plugins.vcs.Git method), 79
- `__iter__()` (compapp.base.DictObject method), 94
- `__len__()` (compapp.base.DictObject method), 94
- `__module__` (compapp.apps.Computer attribute), 90
- `__module__` (compapp.apps.Memoizer attribute), 92
- `__module__` (compapp.base.DictObject attribute), 94
- `__module__` (compapp.base.MultiException attribute), 93
- `__module__` (compapp.core.DataDescriptor attribute), 97
- `__module__` (compapp.core.Defer attribute), 100
- `__module__` (compapp.core.Descriptor attribute), 97
- `__module__` (compapp.core.Parameter attribute), 96
- `__module__` (compapp.core.Parametric attribute), 99
- `__module__` (compapp.core.Private attribute), 96
- `__module__` (compapp.core.WeakRefProperty attribute), 96
- `__module__` (compapp.descriptors.dynamic_class.ClassPath attribute), 37

`__module__` (compapp.descriptors.dynamic_class.ClassPlaceholder attribute), 37

`__module__` (compapp.descriptors.links.Delegate attribute), 40

`__module__` (compapp.descriptors.links.Link attribute), 39

`__module__` (compapp.descriptors.links.MyName attribute), 40

`__module__` (compapp.descriptors.links.OwnerName attribute), 40

`__module__` (compapp.descriptors.links.Root attribute), 39

`__module__` (compapp.descriptors.misc.Constant attribute), 41

`__module__` (compapp.descriptors.tests.test_complex.Nestable attribute), 32

`__module__` (compapp.descriptors.tests.test_complex.Pluggy attribute), 32

`__module__` (compapp.descriptors.tests.test_complex.Root attribute), 33

`__module__` (compapp.descriptors.tests.test_complex.Root.sub attribute), 33

`__module__` (compapp.descriptors.tests.test_complex.Root.sub.sub attribute), 33

`__module__` (compapp.descriptors.tests.test_complex.Root.sub.sub.sub attribute), 33

`__module__` (compapp.descriptors.tests.test_complex.Root.sub.sub.sub.sub attribute), 33

`__module__` (compapp.descriptors.tests.test_or.ParOrDelegate attribute), 34

`__module__` (compapp.descriptors.tests.test_or.ParOrDelegate.maybe attribute), 34

`__module__` (compapp.descriptors.tests.test_params.TestDictWithDefault attribute), 35

`__module__` (compapp.descriptors.tests.test_params.TestDictWithDefault(MyApp attribute), 35

`__module__` (compapp.descriptors.tests.test_params.TestOrDefault attribute), 36

`__module__` (compapp.descriptors.tests.test_params.TestOrDefault(MyApp attribute), 36

`__module__` (compapp.descriptors.traits.Choice attribute), 45

`__module__` (compapp.descriptors.traits.Dict attribute), 44

`__module__` (compapp.descriptors.traits.List attribute), 43

`__module__` (compapp.descriptors.traits.OfType attribute), 42

`__module__` (compapp.descriptors.traits.Optional attribute), 45

`__module__` (compapp.descriptors.traits.Or attribute), 46

`__module__` (compapp.descriptors.traits.Required attribute), 42

`__module__` (compapp.executables.Assembler attribute),

`__module__` (compapp.executables.Assembler.figure attribute), 103

`__module__` (compapp.executables.Loader attribute), 104

`__module__` (compapp.executables.Loader.magics attribute), 104

`__module__` (compapp.executables.MagicPlugins attribute), 101

`__module__` (compapp.executables.Plotter attribute), 104

`__module__` (compapp.interface.Executable attribute), 106

`__module__` (compapp.interface.Plugin attribute), 107

`__module__` (compapp.parser.Option attribute), 108

`__module__` (compapp.plugins.datastores.BaseDataStore attribute), 52

`__module__` (compapp.plugins.datastores.DirectoryDataStore attribute), 54

`__module__` (compapp.plugins.datastores.HashDataStore attribute), 56

`__module__` (compapp.plugins.datastores.SubDataStore attribute), 56

`__module__` (compapp.plugins.metastore.MetaStore attribute), 58

`__module__` (compapp.plugins.misc.AutoUpstreams attribute), 71

`__module__` (compapp.plugins.misc.Debug attribute), 68

`__module__` (compapp.plugins.misc.DebugNS attribute), 66

`__module__` (compapp.plugins.misc.Figure attribute), 71

`__module__` (compapp.plugins.misc.Logger attribute), 66

`__module__` (compapp.plugins.misc.PluginWrapper attribute), 71

`__module__` (compapp.plugins.programinfo.RecordProgramInfo attribute), 73

`__module__` (compapp.plugins.recorders.DumpParameters attribute), 76

`__module__` (compapp.plugins.recorders.DumpResults attribute), 75

`__module__` (compapp.plugins.sysinfo.RecordSysInfo attribute), 77

`__module__` (compapp.plugins.tests.test_autoupstreams.AlwaysReady attribute), 47

`__module__` (compapp.plugins.tests.test_autoupstreams.DependsOnAlways attribute), 48

`__module__` (compapp.plugins.tests.test_autoupstreams.DependsOnDepAR attribute), 49

`__module__` (compapp.plugins.tests.test_autoupstreams.DependsOnTwo attribute), 50

`__module__` (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 51

`__module__` (compapp.plugins.tests.test_datastores.CheckDir attribute), 51

`__module__` (compapp.plugins.tests.test_datastores.CheckDir.sub attribute), 51

__module__ (compapp.plugins.tests.test_datastores.CheckDir.submodule attribute), 51	__module__ (compapp.tests.test_parametric.TestMixObject.AutoMixed attribute), 86
__module__ (compapp.plugins.tests.test_datastores.CheckDir.submodule attribute), 51	__module__ (compapp.tests.test_parametric.TestMixObject.AutoMixed.nes attribute), 86
__module__ (compapp.plugins.tests.test_datastores.WithStore module attribute), 51	__module__ (compapp.tests.test_setters.ClassA attribute), 87
__module__ (compapp.plugins.tests.test_logger.Greeter attribute), 52	__module__ (compapp.tests.test_setters.ClassB attribute), 87
__module__ (compapp.plugins.tests.test_logger.MyApp attribute), 52	__module__ (compapp.tests.test_setters.MyApp attribute), 87
__module__ (compapp.plugins.tests.test_logger.MyApp.sub attribute), 52	__module__ (compapp.variator.ParamBuilder attribute), 111
__module__ (compapp.plugins.tests.test_logger.MyApp.sub attribute), 52	__module__ (compapp.variator.Variator attribute), 113
__module__ (compapp.plugins.timing.RecordTiming attribute), 79	__new__() (compapp.parser.Option static method), 108
__module__ (compapp.plugins.vcs.Git attribute), 79	__repr__() (compapp.base.DictObject method), 94
__module__ (compapp.plugins.vcs.RecordVCS attribute), 81	__repr__() (compapp.parser.Option method), 108
__module__ (compapp.samples.named_figure.MyApp attribute), 81	__set__() (compapp.core.DataDescriptor method), 97
__module__ (compapp.samples.pluggable_plotter_exec.CumHistPlotter attribute), 81	__set__() (compapp.core.WeakRefProperty method), 96
__module__ (compapp.samples.pluggable_plotter_exec.HistPlotter attribute), 81	__set__() (compapp.descriptors.dynamic_class.ClassPlaceholder method), 37
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__set__() (compapp.descriptors.misc.Constant method), 49
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__setattr__() (compapp.core.Parametric method), 98
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__setattr__() (compapp.plugins.misc.DebugNS method), 66
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__setattr__() (compapp.strict.MixinStrict method), 109
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__setitem__() (compapp.base.DictObject method), 94
__module__ (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82	__simulator__ (compapp.parser.Option attribute), 108
__module__ (compapp.samples.pluggable_plotter_link.CumHistPlotter attribute), 83	__str__() (compapp.base.MultiException method), 92
__module__ (compapp.samples.pluggable_plotter_link.HistPlotter attribute), 83	__str__() (compapp.base.DictObject attribute), 94
__module__ (compapp.samples.pluggable_plotter_link.HistPlotter attribute), 83	__weakref__ (compapp.base.MultiException attribute), 93
__module__ (compapp.samples.pluggable_plotter_link.HistPlotter attribute), 83	__weakref__ (compapp.core.Defer attribute), 100
__module__ (compapp.samples.pluggable_plotter_link.MyApp attribute), 84	__weakref__ (compapp.core.Descriptor attribute), 97
__module__ (compapp.samples.pluggable_plotter_link.MyApp attribute), 84	__weakref__ (compapp.core.Parameter attribute), 96
__module__ (compapp.samples.pluggable_plotter_link.MyApp attribute), 84	__weakref__ (compapp.core.Private attribute), 96
__module__ (compapp.samples.pluggable_plotter_link.MyApp attribute), 84	__weakref__ (compapp.core.WeakRefProperty attribute), 96
__module__ (compapp.samples.pluggable_plotter_link.MyApp attribute), 84	__weakref__ (compapp.descriptors.tests.test_params.TestDictWithDefault attribute), 35
__module__ (compapp.strict.MixinStrict attribute), 109	__weakref__ (compapp.plugins.misc.DebugNS attribute), 66
__module__ (compapp.tests.test_core.ParWithMissingLink attribute), 85	__weakref__ (compapp.plugins.vcs.Git attribute), 79
__module__ (compapp.tests.test_parametric.MyParametric attribute), 86	__weakref__ (compapp.strict.MixinStrict attribute), 109
__module__ (compapp.tests.test_parametric.MyParametric.nes attribute), 86	__weakref__ (compapp.tests.test_parametric.TestMixObject.AutoMixed.nes attribute), 86
__module__ (compapp.tests.test_parametric.TestMixClassObj attribute), 87	
__module__ (compapp.tests.test_parametric.TestMixClassObj attribute), 87	
__module__ (compapp.tests.test_parametric.TestMixClassObj attribute), 87	
__module__ (compapp.tests.test_parametric.TestMixClassObj attribute), 87	
__module__ (compapp.tests.test_parametric.TestMixObject attribute), 86	

- allowed (compapp.descriptors.traits.OfType attribute), 42
- AlwaysReady (class in compapp.plugins.tests.test_autoupstreams), 47
- alwaysready (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 50
- alwaysready_done (compapp.plugins.tests.test_autoupstreams.DependsOnAlwaysReady attribute), 47
- app, 21
- application, 21
- argrange (compapp.executables.Assembler attribute), 103
- asdesc() (in module compapp.descriptors.traits), 43
- Assembler (class in compapp.executables), 101
- Assembler.figure (class in compapp.executables), 102
- assert_fallbacks_are() (in module compapp.descriptors.tests.test_complex), 33
- assert_levels() (in module compapp.plugins.tests.test_logger), 52
- assign_option() (in module compapp.parser), 108
- attr (compapp.descriptors.tests.test_or.ParOrDelegate attribute), 33
- attr (compapp.descriptors.tests.test_or.ParOrDelegate.nested attribute), 33
- autoclose (compapp.plugins.misc.Figure attribute), 70
- automixin() (in module compapp.core), 95
- AutoUpstreams (class in compapp.plugins.misc), 71
- autoupstreams (compapp.executables.MagicPlugins attribute), 101
- autoupstreams (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 50
- autoupstreams (compapp.samples.pluggable_plotter_link.MyApp attribute), 83
- ## B
- b (compapp.tests.test_parametric.MyParametric.nested attribute), 86
- base (compapp.variator.Variator attribute), 111
- BaseDataStore (class in compapp.plugins.datastores), 52
- basedir (compapp.plugins.datastores.HashDataStore attribute), 56
- basic_types (in module compapp.core), 95
- bins (compapp.samples.pluggable_plotter_exec.HistPlotter attribute), 81
- bins (compapp.samples.pluggable_plotter_link.HistPlotter attribute), 82
- build_params() (compapp.variator.ParamBuilder method), 111
- builder (compapp.variator.Variator attribute), 111
- ## C
- call() (compapp.core.Defer method), 101
- call_plugins() (in module compapp.interface), 106
- CheckDir (class in compapp.plugins.tests.test_datastores), 51
- CheckDir.sub (class in compapp.plugins.tests.test_datastores), 51
- CheckDir.sub.sub (class in compapp.plugins.tests.test_datastores), 51
- CheckDir.sub.sub.sub (class in compapp.plugins.tests.test_datastores), 51
- choices (compapp.variator.ParamBuilder attribute), 109
- ClassA (class in compapp.tests.test_setters), 87
- ClassB (class in compapp.tests.test_setters), 87
- ClassPath (class in compapp.descriptors.dynamic_class), 36
- classpath (compapp.tests.test_setters.MyApp attribute), 87
- classpath (compapp.variator.Variator attribute), 111
- ClassPlaceholder (class in compapp.descriptors.dynamic_class), 37
- clear_before_run (compapp.plugins.datastores.DirectoryDataStore attribute), 53
- cli() (compapp.apps.Computer method), 90
- cli_mrun() (in module compapp.cli), 95
- cli_run() (in module compapp.cli), 95
- compapp (module), 113
- compapp.apps (module), 89
- compapp.base (module), 92
- compapp.cli (module), 95
- compapp.core (module), 95
- compapp.descriptors (module), 47
- compapp.descriptors.dynamic_class (module), 36
- compapp.descriptors.links (module), 38
- compapp.descriptors.misc (module), 40
- compapp.descriptors.tests (module), 36
- compapp.descriptors.tests.test_complex (module), 31
- compapp.descriptors.tests.test_or (module), 33
- compapp.descriptors.tests.test_params (module), 34
- compapp.descriptors.traits (module), 41
- compapp.executables (module), 101
- compapp.interactive (module), 104
- compapp.interface (module), 104
- compapp.loader (module), 107
- compapp.parser (module), 107
- compapp.plugins (module), 81
- compapp.plugins.datastores (module), 52
- compapp.plugins.metastore (module), 57
- compapp.plugins.misc (module), 58
- compapp.plugins.programinfo (module), 72
- compapp.plugins.recorders (module), 73
- compapp.plugins.sysinfo (module), 76
- compapp.plugins.tests (module), 52
- compapp.plugins.tests.test_autoupstreams (module), 47
- compapp.plugins.tests.test_datastores (module), 51
- compapp.plugins.tests.test_logger (module), 51
- compapp.plugins.timing (module), 77

- compapp.plugins.vcs (module), 79
 - compapp.samples (module), 84
 - compapp.samples.named_figure (module), 81
 - compapp.samples.pluggable_plotter_exec (module), 81
 - compapp.samples.pluggable_plotter_link (module), 82
 - compapp.samples.simple_plots (module), 84
 - compapp.setters (module), 108
 - compapp.strict (module), 109
 - compapp.tests (module), 88
 - compapp.tests.test_base (module), 84
 - compapp.tests.test_core (module), 84
 - compapp.tests.test_loader (module), 86
 - compapp.tests.test_parametric (module), 86
 - compapp.tests.test_setters (module), 87
 - compapp.utils (module), 89
 - compapp.utils.files (module), 88
 - compapp.utils.importer (module), 88
 - compapp.variator (module), 109
 - composition over inheritance, 30
 - Computer (class in compapp.apps), 90
 - configurator (compapp.plugins.misc.Logger attribute), 59
 - configure() (compapp.plugins.misc.Logger method), 66
 - Constant (class in compapp.descriptors.misc), 40
 - constant() (in module compapp.base), 92
 - countprefix() (in module compapp.descriptors.links), 38
 - critical (compapp.plugins.misc.Logger attribute), 58, 60
 - cumdist (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82
 - cumdist (compapp.samples.pluggable_plotter_link.MyApp attribute), 84
 - CumHistPlotter (class in compapp.samples.pluggable_plotter_exec), 81
 - CumHistPlotter (class in compapp.samples.pluggable_plotter_link), 83
 - cumulative (compapp.samples.pluggable_plotter_exec.CumHistPlotter attribute), 81
 - cumulative (compapp.samples.pluggable_plotter_link.CumHistPlotter attribute), 83
- ## D
- dassert_fallbacks_are() (in module compapp.descriptors.tests.test_complex), 33
 - data (compapp.samples.pluggable_plotter_link.HistPlotter attribute), 82
 - data sink, 21
 - data source, 21
 - DataDescriptor (class in compapp.core), 97
 - datastore, 29
 - datastore (compapp.apps.Memoizer attribute), 92
 - datastore (compapp.executables.Assembler attribute), 103
 - datastore (compapp.executables.Assembler.figure attribute), 103
 - datastore (compapp.executables.Loader attribute), 104
 - datastore (compapp.executables.Plotter attribute), 104
 - datastore (compapp.plugins.metastore.MetaStore attribute), 57
 - datastore (compapp.plugins.misc.Figure attribute), 69
 - datastore (compapp.plugins.misc.Logger attribute), 65
 - datastore (compapp.plugins.tests.test_datastores.WithStore attribute), 51
 - datastore_format (compapp.variator.Variator attribute), 112
 - dbg (compapp.executables.Assembler attribute), 102
 - Debug (class in compapp.plugins.misc), 66
 - debug (compapp.executables.Assembler attribute), 103
 - debug (compapp.plugins.misc.Logger attribute), 58, 64
 - DebugNS (class in compapp.plugins.misc), 66
 - deepmixdicts() (in module compapp.base), 94
 - default (compapp.descriptors.traits.Or attribute), 47
 - defaultparams() (compapp.core.Parametric class method), 99
 - Defer (class in compapp.core), 100
 - defer (compapp.interface.Executable attribute), 105
 - Delegate (class in compapp.descriptors.links), 39
 - density (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82
 - density (compapp.samples.pluggable_plotter_link.MyApp attribute), 83
 - depar (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 50
 - depar_done (compapp.plugins.tests.test_autoupstreams.DependsOnDepAR attribute), 48
 - depdepar (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 50
 - depdepar_done (compapp.plugins.tests.test_autoupstreams.DependsOnTwo attribute), 49
 - DependsOnAlwaysReady (class in compapp.plugins.tests.test_autoupstreams), 47
 - DependsOnDepAR (class in compapp.plugins.tests.test_autoupstreams), 48
 - DependsOnTwo (class in compapp.plugins.tests.test_autoupstreams), 49
 - deptwo (compapp.plugins.tests.test_autoupstreams.RootApp attribute), 50
 - Descriptor (class in compapp.core), 96
 - Dict (class in compapp.descriptors.traits), 43
 - DictObject (class in compapp.base), 93
 - dir (compapp.plugins.datastores.DirectoryDataStore attribute), 53
 - dir (compapp.plugins.datastores.SubDataStore attribute), 55
 - DirectoryDataStore (class in compapp.plugins.datastores), 53
 - dotted_to_nested() (in module compapp.base), 94
 - DumpParameters (class in compapp.plugins.recorders), 75
 - dumppparameters (compapp.executables.MagicPlugins attribute), 75

tribute), 101
DumpResults (class in compapp.plugins.recorders), 73
dumpresults (compapp.executables.Loader.magics attribute), 104
dumpresults (compapp.executables.MagicPlugins attribute), 101
dynamic (compapp.tests.test_setters.MyApp attribute), 87
dynamic_class() (in module compapp.descriptors.dynamic_class), 37

E

enable (compapp.plugins.misc.Debug attribute), 68
error (compapp.plugins.misc.Logger attribute), 58, 61
Executable (class in compapp.interface), 104
execute() (compapp.interface.Executable method), 106
execute() (in module compapp.variator), 111
executor (compapp.variator.Variator attribute), 111
exists (compapp.plugins.tests.test_datastores.WithStore attribute), 51
exists() (compapp.plugins.datastores.DirectoryDataStore method), 54
ext (compapp.plugins.misc.Figure attribute), 69

F

fallback (compapp.descriptors.tests.test_complex.Pluggy attribute), 31
Figure (class in compapp.plugins.misc), 68
figure (compapp.executables.Plotter attribute), 104
finish() (compapp.interface.Executable method), 106
finish() (compapp.interface.Plugin method), 107
finish() (compapp.plugins.misc.Figure method), 70
finish() (compapp.plugins.misc.PluginWrapper method), 71
formatters (compapp.plugins.misc.Logger attribute), 60
from_param() (compapp.apps.Computer class method), 90

G

get() (compapp.core.DataDescriptor method), 97
get() (compapp.descriptors.dynamic_class.ClassPlaceholderis method), 37
get() (compapp.descriptors.links.Delegate method), 40
get() (compapp.descriptors.links.Link method), 39
get() (compapp.descriptors.links.MyName method), 40
get() (compapp.descriptors.links.OwnerName method), 40
get() (compapp.descriptors.misc.Constant method), 40
get() (compapp.descriptors.traits.OfType method), 42
get() (compapp.descriptors.traits.Or method), 46
get_parser() (compapp.apps.Computer class method), 90
getclass() (compapp.descriptors.dynamic_class.ClassPath method), 37
getroot() (compapp.core.Private method), 96

getrusage_self() (in module compapp.plugins.timing), 77
gettimesteps() (in module compapp.plugins.timing), 77
getvcs() (in module compapp.plugins.vcs), 79
Git (class in compapp.plugins.vcs), 79
git() (compapp.plugins.vcs.Git method), 79
git_cmd (compapp.plugins.vcs.Git attribute), 79
Global Figure plugin configuration, 69
globitems() (compapp.plugins.datastores.DirectoryDataStore method), 54
globitems() (compapp.plugins.datastores.SubDataStore method), 56
Greeter (class in compapp.plugins.tests.test_logger), 51

H

handlers (compapp.plugins.misc.Logger attribute), 60
has_required_attributes() (in module compapp.descriptors.traits), 43
HashDataStore (class in compapp.plugins.datastores), 56
hexdigest() (in module compapp.plugins.datastores), 56
HistPlotter (class in compapp.samples.pluggable_plotter_exec), 81
HistPlotter (class in compapp.samples.pluggable_plotter_link), 82

I

i (compapp.tests.test_parametric.MyParametric attribute), 86
import_appclass() (in module compapp.cli), 95
import_object() (in module compapp.utils.importer), 88
info (compapp.plugins.misc.Logger attribute), 58, 63
is_enabled() (compapp.plugins.misc.Debug method), 68
is_loadable (compapp.plugins.datastores.DirectoryDataStore attribute), 54
is_loadable() (compapp.executables.Assembler method), 103
is_logging_debug() (compapp.plugins.misc.Debug method), 68
is_runnable() (compapp.plugins.misc.AutoUpstreams static method), 71
is_runnable() (in module compapp.plugins.misc), 71
is_writable() (compapp.plugins.datastores.DirectoryDataStore method), 54
isclean() (compapp.plugins.vcs.Git method), 79
isparam (compapp.core.DataDescriptor attribute), 97
isparam (compapp.core.Descriptor attribute), 96
iswritable() (in module compapp.plugins.datastores), 52
itervars() (in module compapp.base), 95

K

key (compapp.descriptors.traits.Or attribute), 46
keyed() (compapp.core.Defer method), 101
keys() (compapp.variator.ParamBuilder method), 111

L

level (compapp.plugins.misc.Logger attribute), 60
 lhs (compapp.parser.Option attribute), 108
 Link (class in compapp.descriptors.links), 38
 link (compapp.tests.test_core.ParWithMissingLink attribute), 84
 linspaces (compapp.variator.ParamBuilder attribute), 110
 List (class in compapp.descriptors.traits), 43
 load() (compapp.interface.Executable method), 106
 load() (compapp.interface.Plugin method), 107
 load() (compapp.plugins.metastore.MetaStore method), 58
 load() (compapp.plugins.misc.PluginWrapper method), 71
 load() (compapp.plugins.recorders.DumpParameters method), 76
 load() (compapp.plugins.recorders.DumpResults method), 75
 load() (in module compapp.loader), 107
 load_any() (in module compapp.parser), 108
 load_results_hdf5() (compapp.plugins.recorders.DumpResults method), 75
 load_results_json() (compapp.plugins.recorders.DumpResults method), 75
 load_results_npz() (compapp.plugins.recorders.DumpResults method), 75
 Loader (class in compapp.executables), 103
 Loader.magics (class in compapp.executables), 103
 log (compapp.executables.Assembler attribute), 103
 log (compapp.plugins.metastore.MetaStore attribute), 57
 log (compapp.plugins.misc.Debug attribute), 67
 log (compapp.plugins.misc.Logger attribute), 58
 Logger (class in compapp.plugins.misc), 58
 logspaces (compapp.variator.ParamBuilder attribute), 110

M

MagicPlugins (class in compapp.executables), 101
 magics (compapp.executables.Assembler attribute), 101
 main() (in module compapp.cli), 95
 make_parser() (in module compapp.cli), 95
 make_parser() (in module compapp.parser), 108
 makemethod() (in module compapp.plugins.misc), 71
 Memoizer (class in compapp.apps), 90
 message (compapp.plugins.tests.test_logger.Greeter attribute), 52
 meta (compapp.executables.MagicPlugins attribute), 101
 meta (compapp.plugins.programinfo.RecordProgramInfo attribute), 72
 meta (compapp.plugins.sysinfo.RecordSysInfo attribute), 76

meta (compapp.plugins.timing.RecordTiming attribute), 78
 meta (compapp.plugins.vcs.RecordVCS attribute), 80
 metafile (compapp.plugins.metastore.MetaStore attribute), 57
 metafilepath (compapp.plugins.metastore.MetaStore attribute), 58
 MetaStore (class in compapp.plugins.metastore), 57
 mixdicts() (in module compapp.base), 95
 MixinStrict (class in compapp.strict), 109
 mode (compapp.apps.Memoizer attribute), 91
 mode (compapp.executables.Assembler attribute), 101
 modifier (compapp.parser.Option attribute), 108
 MultiException, 92
 MyApp (class in compapp.plugins.tests.test_logger), 52
 MyApp (class in compapp.samples.named_figure), 81
 MyApp (class in compapp.samples.pluggable_plotter_exec), 82
 MyApp (class in compapp.samples.pluggable_plotter_link), 83
 MyApp (class in compapp.samples.simple_plots), 84
 MyApp (class in compapp.tests.test_setters), 87
 MyApp.sub (class in compapp.plugins.tests.test_logger), 52
 MyApp.sub.sub (class in compapp.plugins.tests.test_logger), 52
 MyName (class in compapp.descriptors.links), 40
 myname (compapp.descriptors.traits.Or attribute), 46
 myname() (compapp.core.Descriptor method), 96
 MyParametric (class in compapp.tests.test_parametric), 86
 MyParametric.nested (class in compapp.tests.test_parametric), 86
 MySimulator (class in compapp.samples.pluggable_plotter_exec), 81
 MySimulator (class in compapp.samples.pluggable_plotter_link), 83

N

nametemplate (compapp.plugins.misc.Logger attribute), 65
 Nestable (class in compapp.descriptors.tests.test_complex), 32
 nested class, 29

O

OfType (class in compapp.descriptors.traits), 41
 on (compapp.plugins.datastores.DirectoryDataStore attribute), 53
 Option (class in compapp.parser), 107
 Optional (class in compapp.descriptors.traits), 44
 Or (class in compapp.descriptors.traits), 45
 overwrite (compapp.plugins.datastores.DirectoryDataStore attribute), 53

ownconfig (compapp.plugins.misc.Logger attribute), 60
owner, 29
owner (compapp.core.Private attribute), 96
owner app, 29
owner class, 29
ownerhash() (compapp.plugins.datastores.HashDataStore method), 56
OwnerName (class in compapp.descriptors.links), 40

P

ParamBuilder (class in compapp.variator), 109
Parameter (class in compapp.core), 96
Parametric (class in compapp.core), 97
paramnames() (compapp.core.Parametric class method), 99
params() (compapp.core.Parametric method), 98
ParOrDelegate (class in compapp.descriptors.tests.test_or), 33
ParOrDelegate.nested (class in compapp.descriptors.tests.test_or), 33
parse() (compapp.descriptors.traits.Choice method), 45
parse() (compapp.descriptors.traits.OfType method), 42
parse() (compapp.descriptors.traits.Or method), 46
parse_assignment_options() (in module compapp.parser), 108
parse_bool() (in module compapp.parser), 108
parse_key() (in module compapp.parser), 108
parse_value() (in module compapp.parser), 108
parseargs() (in module compapp.parser), 108
ParWithMissingLink (class in compapp.tests.test_core), 84
path() (compapp.plugins.datastores.DirectoryDataStore method), 54
path() (compapp.plugins.datastores.SubDataStore method), 56
Plotter (class in compapp.executables), 104
Pluggy (class in compapp.descriptors.tests.test_complex), 31
pluggy (compapp.descriptors.tests.test_complex.Nestable attribute), 32
Plugin (class in compapp.interface), 107
PluginWrapper (class in compapp.plugins.misc), 71
post_run() (compapp.interface.Plugin method), 107
post_run() (compapp.plugins.misc.PluginWrapper method), 71
post_run() (compapp.plugins.timing.RecordTiming method), 79
pre_run() (compapp.interface.Plugin method), 107
pre_run() (compapp.plugins.misc.PluginWrapper method), 71
pre_run() (compapp.plugins.programinfo.RecordProgramInfo method), 73
pre_run() (compapp.plugins.recorders.DumpParameters method), 76

pre_run() (compapp.plugins.recorders.DumpResults method), 74
pre_run() (compapp.plugins.sysinfo.RecordSysInfo method), 77
pre_run() (compapp.plugins.timing.RecordTiming method), 79
pre_run() (compapp.plugins.vcs.RecordVCS method), 81
prepare() (compapp.interface.Executable method), 106
prepare() (compapp.interface.Plugin method), 107
prepare() (compapp.plugins.datastores.DirectoryDataStore method), 54
prepare() (compapp.plugins.datastores.HashDataStore method), 56
prepare() (compapp.plugins.metastore.MetaStore method), 58
prepare() (compapp.plugins.misc.AutoUpstreams method), 71
prepare() (compapp.plugins.misc.Figure method), 70
prepare() (compapp.plugins.misc.Logger method), 66
prepare() (compapp.plugins.misc.PluginWrapper method), 71
print_full_help() (in module compapp.apps), 89
Private (class in compapp.core), 96
private() (in module compapp.core), 96
process_assignment_options() (in module compapp.parser), 108
process_modifier() (in module compapp.parser), 108
processes (compapp.variator.Variator attribute), 111
programinfo (compapp.executables.MagicPlugins attribute), 101
props_of() (in module compapp.base), 95

R

ranges (compapp.variator.ParamBuilder attribute), 110
real_owner() (in module compapp.plugins.misc), 71
rec_setattr() (in module compapp.setters), 108
rec_setattrs() (in module compapp.setters), 108
record() (compapp.base.MultiException method), 92
record() (compapp.plugins.metastore.MetaStore method), 58
recorder() (compapp.base.MultiException class method), 92
RecordProgramInfo (class in compapp.plugins.programinfo), 72
RecordSysInfo (class in compapp.plugins.sysinfo), 76
RecordTiming (class in compapp.plugins.timing), 77
recordtiming (compapp.executables.MagicPlugins attribute), 101
RecordVCS (class in compapp.plugins.vcs), 79
recordvcs (compapp.executables.MagicPlugins attribute), 101
register() (compapp.plugins.misc.Figure method), 70
Required (class in compapp.descriptors.traits), 42

result_names (compapp.plugins.recorders.DumpResults attribute), 74
 results (compapp.executables.Assembler attribute), 101
 revision() (compapp.plugins.vcs.Git method), 79
 rhs (compapp.parser.Option attribute), 108
 Root (class in compapp.descriptors.links), 39
 Root (class in compapp.descriptors.tests.test_complex), 32
 root (compapp.plugins.sysinfo.RecordSysInfo attribute), 77
 Root.sub (class in compapp.descriptors.tests.test_complex), 32
 Root.sub.sub (class in compapp.descriptors.tests.test_complex), 32
 Root.sub.sub.sub (class in compapp.descriptors.tests.test_complex), 32
 Root.sub.sub.sub.sub (class in compapp.descriptors.tests.test_complex), 32
 RootApp (class in compapp.plugins.tests.test_autoupstreams), 50
 run() (compapp.base.MultiException class method), 93
 run() (compapp.interface.Executable method), 106
 run() (compapp.plugins.tests.test_autoupstreams.AlwaysReady method), 47
 run() (compapp.plugins.tests.test_datastores.CheckDir method), 51
 run() (compapp.plugins.tests.test_datastores.WithStore method), 51
 run() (compapp.plugins.tests.test_logger.Greeter method), 52
 run() (compapp.samples.named_figure.MyApp method), 81
 run() (compapp.samples.pluggable_plotter_exec.HistPlotter method), 81
 run() (compapp.samples.pluggable_plotter_exec.MyApp method), 82
 run() (compapp.samples.pluggable_plotter_exec.MySimulator method), 82
 run() (compapp.samples.pluggable_plotter_link.HistPlotter method), 83
 run() (compapp.samples.pluggable_plotter_link.MySimulator method), 83
 run() (compapp.samples.simple_plots.MyApp method), 84
 run() (compapp.variator.Variator method), 113

save() (compapp.plugins.misc.Figure method), 70
 save() (compapp.plugins.misc.PluginWrapper method), 71
 save() (compapp.plugins.recorders.DumpResults method), 74
 save_maybe() (compapp.plugins.recorders.DumpResults method), 75
 save_now() (compapp.plugins.recorders.DumpResults method), 74
 save_results() (compapp.plugins.recorders.DumpResults class method), 74
 save_results_hdf5() (compapp.plugins.recorders.DumpResults static method), 74
 save_results_json() (compapp.plugins.recorders.DumpResults static method), 74
 save_results_npz() (compapp.plugins.recorders.DumpResults static method), 74
 sep (compapp.plugins.datastores.SubDataStore attribute), 56
 setup_context() (compapp.core.Private method), 96
 setup_interactive() (in module compapp.interactive), 104
 should_configure() (compapp.plugins.misc.Logger method), 66
 should_load() (compapp.executables.Assembler method), 103
 should_load() (compapp.interface.Executable method), 106
 show (compapp.plugins.misc.Figure attribute), 70
 sim (compapp.samples.pluggable_plotter_exec.MyApp attribute), 82
 sim (compapp.samples.pluggable_plotter_link.MyApp attribute), 83
 simple_type_check() (in module compapp.core), 95
 skip_non_str() (in module compapp.descriptors.traits), 41
 sub (compapp.tests.test_core.ParWithMissingLink attribute), 84
 SubDataStore (class in compapp.plugins.datastores), 54
 subplots() (compapp.plugins.misc.Figure method), 71
 sysinfo (compapp.executables.MagicPlugins attribute), 101

T

TBE, 29

test() (compapp.descriptors.tests.test_params.TestDictWithDefault method), 35
 test_assembler_init() (in module compapp.tests.test_parametric), 87
 test_custom_value() (compapp.tests.test_parametric.TestMixObject method), 86

S

safewrite() (in module compapp.utils.files), 88
 samples (compapp.samples.pluggable_plotter_exec.MySimulator attribute), 81
 samples (compapp.samples.pluggable_plotter_link.MySimulator attribute), 83
 save() (compapp.interface.Executable method), 106
 save() (compapp.interface.Plugin method), 107

test_datastore_log() (in module compapp.plugins.tests.test_logger), 52

test_default_value() (in module compapp.tests.test_parametric.TestMixObject method), 86

test_dict_wo_default() (in module compapp.descriptors.tests.test_params), 34

test_execute() (in module compapp.plugins.tests.test_autoupstreams), 51

test_formatters_are_copied() (in module compapp.plugins.tests.test_logger), 51

test_function() (in module compapp.descriptors.tests.test_or), 34

test_handlers_are_copied() (in module compapp.plugins.tests.test_logger), 51

test_is_runnable_simple() (in module compapp.plugins.tests.test_autoupstreams), 51

test_key_propagation() (in module compapp.descriptors.tests.test_or), 34

test_link_default() (in module compapp.descriptors.tests.test_params), 36

test_list_wo_default() (in module compapp.descriptors.tests.test_params), 34

test_load_dynamic_class() (in module compapp.tests.test_loader), 86

test_missing_link() (in module compapp.tests.test_core), 85

test_myname_propagation() (in module compapp.descriptors.tests.test_or), 34

test_nested_root_config() (in module compapp.plugins.tests.test_logger), 52

test_nested_simple_run() (in module compapp.plugins.tests.test_logger), 52

test_nested_specific_handler() (in module compapp.plugins.tests.test_logger), 52

test_nodir() (in module compapp.plugins.tests.test_datastores), 51

test_propagated_myname() (in module compapp.descriptors.tests.test_or), 34

test_propagation() (in module compapp.descriptors.tests.test_complex), 33

test_root_exists() (in module compapp.plugins.tests.test_datastores), 51

test_sub_exists() (in module compapp.plugins.tests.test_datastores), 51

test_subs_dynamic_class() (in module compapp.tests.test_setters), 87

test_unspecified_bool() (in module compapp.tests.test_base), 84

test_unspecified_pickleable() (in module compapp.tests.test_base), 84

test_unspecified_repr() (in module compapp.tests.test_base), 84

TestDictWithDefault (class in compapp.descriptors.tests.test_params), 34

TestDictWithDefault.MyApp (class in compapp.descriptors.tests.test_params), 34

TestMixClassObj (class in compapp.tests.test_parametric), 86

TestMixClassObj.AutoMixed (class in compapp.tests.test_parametric), 86

TestMixClassObj.AutoMixed.nested (class in compapp.tests.test_parametric), 86

TestMixObject (class in compapp.tests.test_parametric), 86

TestMixObject.AutoMixed (class in compapp.tests.test_parametric), 86

TestMixObject.AutoMixed.nested (class in compapp.tests.test_parametric), 86

TestOrDefault (class in compapp.descriptors.tests.test_params), 35

TestOrDefault.MyApp (class in compapp.descriptors.tests.test_params), 35

to be extended, 30

tupleoftypes() (in module compapp.descriptors.traits), 41

V

variants (compapp.variator.Variator attribute), 112

Variator (class in compapp.variator), 111

vcsinfo() (compapp.plugins.vcs.Git method), 79

vcstype (compapp.plugins.vcs.Git attribute), 79

verify() (compapp.core.DataDescriptor method), 97

verify() (compapp.descriptors.dynamic_class.ClassPath method), 37

verify() (compapp.descriptors.traits.Choice method), 45

verify() (compapp.descriptors.traits.Dict method), 44

verify() (compapp.descriptors.traits.List method), 43

verify() (compapp.descriptors.traits.OfType method), 42

verify() (compapp.descriptors.traits.Optional method), 45

verify() (compapp.descriptors.traits.Or method), 46

W

warn (compapp.plugins.misc.Logger attribute), 58, 62

WeakRefProperty (class in compapp.core), 95

WithStore (class in compapp.plugins.tests.test_datastores), 51

X

x (compapp.descriptors.tests.test_params.TestDictWithDefault.MyApp attribute), 34

x (compapp.descriptors.tests.test_params.TestOrDefault.MyApp attribute), 35

x (compapp.tests.test_parametric.MyParametric attribute), 86

x (compapp.tests.test_setters.ClassA attribute), 87

x (compapp.tests.test_setters.ClassB attribute), 87