Persist Documentation

Release 0.9b1

Michael McNeil Forbes

May 11, 2021

Contents

Persistent archival of python objects in an importable format.

This package provides a method for archiving python objects to disk for long-term persistent storage. The archives are importable python packages with large data stored in the npy numpy data format, or HDF5 files using the h5py package (if it is installed). The original goal was to overcomes several disadvatages of pickles:

1. Archives are relatively stable to code changes. Unlike pickles, changing the underlying code for a class will not change the ability to read an archive if the API does not change.
2. In the presence of API changes, the archives can be edited by hand to fix them since they are simply python code. (Note: for reliability, the generated code is highly structured and not so "pretty", but can still be edited or debugged in the case of errors due to API changes.)
3. Efficient storage of large arrays.
4. Safe for concurrent access by multiple processes.

Documentation: http://persist.readthedocs.org

Source: https://alum.mit.edu/www/mforbes/hg/forbes-group/persist

Issues: https://alum.mit.edu/www/mforbes/hg/forbes-group/issues

CHAPTER 1

Installing

This package can be installed from PyPI:

```
python3 -m pip install persist
```

or from source:

```
python3 -m pip install hg+https://alum.mit.edu/www/mforbes/hg/forbes-group/persist
```

CHAPTER 2

DataSet Format

Table of Contents

2.1 Archives and DataSets

There are two main classes provided by the `persist` module: `persist.Archive` and `persist.DataSet`.

Archives deal with the linkage between objects so that if multiple objects are referred to, they are only stored once in the archive. Archives provide two main methods for to serialize the data:

1. Via the `str()` operator which will return a string that can be executed to restore the archive.
2. Via the `Archive.save()` method which will export the archive to an importable python package or module.

DataSets use archives to provide storage for multiple sets of data along with associated metadata. Each set of data is designed to be accessed concurrently using locks.

2.2 Archive Format

The `persist.Archive` object maintains a collection of python objects that are inserted with `persist.Archive.insert()`. This can be serialized to a string and then reconstituted through evaluation:

2.2.1 Basic Usage

[1]:
```
from persist.archive import Archive

a = 1
x = range(2)
y = range(3)   # Implicitly reference in archive
b = [x, y, y]  # Nested references to x and y

# scoped=False is prettier, but slower and not as safe
archive = Archive(scoped=False)
archive.insert(a=a, x=x, b=b)

# Get the string representation
s = str(archive)
print(s)
```

```
from builtins import range as _range
_g3 = _range(0, 3)
x = _range(0, 2)
b = [x, _g3, _g3]
a = 1
del _range
del _g3
try: del __builtins__, _arrays
except NameError: pass
```

[2]:
```
d = {}
exec(s, d)
print(d)
assert d['a'] == a
assert d['x'] == x
assert d['b'] == b
assert d['b'][1] is d['b'][2]  # Note: these are the same object
```

```
{'x': range(0, 2), 'b': [range(0, 2), range(0, 3), range(0, 3)], 'a': 1}
```

2.2.2 Large Arrays

If you have large arrays of data, then it is better to store those externally. To do this, set the `array_threshold` to specify the maximum number of elements to store in an inline array. Any large array will be stored in `Archive.data` and not be included in the string representation. To properly reconstitute the archive, this data must be provided in the environment as a dictionary with key `Archive.data_name` which defaults to `_arrays`:

[3]:
```
import os.path, tempfile, shutil, numpy as np
from persist.archive import Archive

a = 1
x = np.arange(10)
y = np.arange(20)  # Implicitly reference in archive
b = [x, y]

archive = Archive(scoped=False, array_threshold=5)
archive.insert(a=a, x=x, b=b)

# Get the string representation
s = str(archive)
print(s)
print(archive.data)
```

```
x = _arrays['array_0']
b = [x, _arrays['array_1']]
a = 1
try: del __builtins__, _arrays
except NameError: pass
{'array_0': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'array_1': array([ 0,  1,  2,  3,␣
↪4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])}
```

To evaluate the string representation, we need to provide the `_arrays` dictionary:

[4]:
```
d = dict(_arrays=archive.data)
exec(s, d)
print(d)
```

```
{'x': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'b': [array([0, 1, 2, 3, 4, 5, 6, 7, 8,␣
↪9]), array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])], 'a': 1}
```

To store the data, use `Archive.save_data()`:

[5]:
```
import os.path, tempfile
tmpdir = tempfile.mkdtemp()  # Make temporary directory for data
datafile = os.path.join(tmpdir, 'arrays')
archive.save_data(datafile=datafile)
print(tmpdir)
!ls $tmpdir/arrays
!rm -rf $tmpdir
```

```
/var/folders/m7/dnr91tjs4gn58_t3k8zp_g000000gp/T/tmp5jutcwlt
array_0.npy array_1.npy
```

2.2.3 Importable Archives

(New in version 1.0)

Archives can be saved as importable packages using the `save()` method. This will write the representable portion of the archive as an importable module with additional code to load any external arrays. Archives can be saved as a full package – a directory with a `<name>/__init__.py` file etc. or as a single `<name>.py` module. These can be imported without the `persist` package:

[6]:
```
import os.path, sys, tempfile, shutil, numpy as np
from persist.archive import Archive

tmpdir = tempfile.mkdtemp()

a = 1
x = np.arange(10)
y = np.arange(20)  # Implicitly reference in archive
b = [x, y]

archive = Archive(array_threshold=5)
archive.insert(a=a, x=x, b=b)
archive.save(dirname=tmpdir, name='mod1', package=True)
archive.save(dirname=tmpdir, name='mod2', package=False)

!tree $tmpdir

sys.path.append(tmpdir)
import mod1, mod2
sys.path.pop()
for mod in [mod1, mod2]:
    assert mod.a == a and np.allclose(mod.x, x)
!rm -rf $tmpdir
```

```
/var/folders/m7/dnr91tjs4gn58_t3k8zp_g000000gp/T/tmp9ahg71yq
|-- mod1
|   |-- __init__.py
|   `-- _arrays
|       |-- array_0.npy
|       `-- array_1.npy
|-- mod2.py
`-- mod2_arrays
    |-- array_0.npy
    `-- array_1.npy

3 directories, 6 files
```

2.3 Archive Details

2.3.1 Single-item Archives

(New in version 1.0)

If an archive contains a single item, then the representation can be simplified so that importing the module will result in the actual object. This is mainly for use in `DataSets` where it allows us to have large objects in a module that only get loaded if explicitly imported. In this case, one can also omit the name when calling `Archive.save()` as it defaults to the name of the single item.

[7]:
```
import os.path, sys, tempfile, shutil, numpy as np
from persist.archive import Archive

tmpdir = tempfile.mkdtemp()

x = np.arange(10)
y = np.arange(20)  # Implicitly reference in archive
```

(continues on next page)

(continued from previous page)

```
b = [x, y]

archive = Archive(single_item_mode=True, array_threshold=5)
archive.insert(b1=b)
archive.save(dirname=tmpdir, package=True)

archive = Archive(scoped=False, single_item_mode=True, array_threshold=5)
archive.insert(b2=b)
archive.save(dirname=tmpdir, package=False)

!tree $tmpdir

sys.path.append(tmpdir)
import b1, b2
sys.path.pop()
for b_ in [b1, b2]:
    assert np.allclose(b_[0], x) and np.allclose(b_[1], y)
!rm -rf $tmpdir
```

```
/var/folders/m7/dnr91tjs4gn58_t3k8zp_g000000gp/T/tmpval74y_i
|-- b1
|   |-- __init__.py
|   `-- _arrays
|       |-- array_0.npy
|       `-- array_1.npy
|-- b2.py
`-- b2_arrays
    |-- array_0.npy
    `-- array_1.npy

3 directories, 6 files
```

Note what is happening here... although we explicitly `import b1`, the result of this is that `b1 = [x, y]` is the list rather than a module. This behaviour is somewhat of an abuse of the import system, so you should not expose it too much. The use in `DataSet` is that these modules are included as submodules of the DataSet package, acting as attributes of the top-level package, but only being loaded when explicitly imported to limit memory usage etc.

2.3.2 Containers, Duplicates, and Circular References

The main complexity with archives is with objects like lists and dictionaries that refer to other objects: all object referenced in such "containers" need to be stored only one time in the archive. A current limitation is that circular dependencies cannot be resolved. The pickling mechanism provides a way to restore circular dependencies, but I do not see an easy way to resolve this in a human-readable format, so the current requirement is that the references in an object form a directed acyclic graph (DAG).

2.4 Archive Examples

Here we demonstrate a simple archive containing all of the data. We start with the simplest format which is obtained with `scoped=False`:

2.4.1 Non-scoped (flat) Format

We start with the `scoped=False` format. This produces a flat archive that is easier to read:

[8]:
```
import os.path, tempfile, shutil
from persist.archive import Archive

a = 1
x = range(2)
y = range(3)   # Implicitly reference in archive
b = [x, y, y]  # Nested references to x and y

archive = Archive(scoped=False)
archive.insert(a=a, x=x, b=b)

# Get the string representation
%time s = str(archive)
print(s)
```

```
CPU times: user 778 µs, sys: 210 µs, total: 988 µs
Wall time: 927 µs
from builtins import range as _range
_g3 = _range(0, 3)
x = _range(0, 2)
b = [x, _g3, _g3]
a = 1
del _range
del _g3
try: del __builtins__, _arrays
except NameError: pass
```

Note that intermediate objects not explicitly inserted are stored with variables like `_g#` and that these are deleted, so that evaluating the string in a dictionary gives a clean result:

[9]:
```
# Now execute the representation to get the data
d = {}
exec(s, d)
print(d)
d['b'][1] is d['b'][2]
```

```
{'x': range(0, 2), 'b': [range(0, 2), range(0, 3), range(0, 3)], 'a': 1}
```

[9]:
```
True
```

The potential problem with the flat format is that to obtain this simple representation, a graph reduction is performed that replaces intermediate nodes, ensuring that local variables do not have name clashes as well as simplifing the representation. Replacing variables in representations can have performance implications if the objects are large. The fastest approach is a string replacement, but this can make mistakes if the substring appears in data. The option `robust_replace` invokes the python AST parser, but this is slower.

Scoped Format

To alleviate these issues, the `scoped=True` format is provided. This is visually much more complicated as each object is constructed in a function. The advantage is that this provides a local scope in which objects are defined. As a result, any local variables defined in the representation of the object can be used as they are without worrying that they will conflict with other names in the file. No reduction is performed and no replacements are made, makeing the method faster and more robust, but less attractive if the files need to be inspected by humans:

[10]:
```
archive = Archive(scoped=True)
archive.insert(a=a, x=x, b=b)

# Get the string representation
%time s = str(archive)
print(s)
```

```
CPU times: user 412 µs, sys: 22 µs, total: 434 µs
Wall time: 452 µs

def _g3():
    from builtins import range
    return range(0, 3)
_g3 = _g3()

def x():
    from builtins import range
    return range(0, 2)
x = x()

def b(_l_0=x,_l_1=_g3,_l_2=_g3):
    return [_l_0, _l_1, _l_2]
b = b()
a = 1
del _g3
try: del __builtins__, _arrays
except NameError: pass
```

2.5 DataSet Format

This is the new format of DataSets starting with revision 1.0.

A DataSet is a directory with the following files:

- `_this_dir_is_a_DataSet`: This is an empty file signifying that the directory is a DataSet.
- `__init__.py`: Each DataSet is an importable python module so that the data can be used on a machine without the `persist` package. This file contains the following variable:
- `_info_dict`: This is a dictionary/namespace with string keys (which must be valid python identifiers) and associated data (which should in general be small). These are intended to be interpreted as meta-data.

For the remainder of this discussion, we shall assume that `_info_dict` contains the key `'x'`.

- `x.py`: This is the python file responsible for loading the data associated with the key `'x'` in `_info_dict`. If the size of the array is less than the `array_threshold` specified in the `DataSet` object, then the data for the arrays are stored in this file, otherwise this file is responsible for loading the data from an associated file.
- `x_data.*`: If the size of the array stored in `x` is larger than the `array_threshold`, then the data associated with `x` is stored in this file/directory which may be an HDF5 file, or a numpy array file.

These DataSet modules can be directly imported. Importing the top-level DataSet will result in a module with the `_info_dict` attribute containing all the meta data. The data items become available when you explicitly import them.

2.6 DataSet Examples

[11]:
```
import os.path, pprint, sys, tempfile, shutil, numpy as np
from persist.archive import DataSet
tmpdir = tempfile.mkdtemp()  # Make temporary directory for dataset
print("Storing dataset in {}".format(tmpdir))

a = np.arange(10)
x = np.arange(15)

ds = DataSet('dataset', 'w', path=tmpdir, array_threshold=12, data_format='npy')

ds.a = a
ds.x = [a, x]
ds['a'] = "A small array"
ds['x'] = "A list with a small and large array"

!tree $tmpdir

del ds

ds = DataSet('dataset', 'r', path=tmpdir)
print(ds['a'])
print(ds['x'])
print(ds.a)   # The arrays a and x are not actually loaded until here
print(ds.x)
```

```
Storing dataset in /var/folders/m7/dnr91tjs4gn58_t3k8zp_g000000gp/T/tmplw345fr6
/var/folders/m7/dnr91tjs4gn58_t3k8zp_g000000gp/T/tmplw345fr6
`-- dataset
    |-- __init__.py
    |-- _this_dir_is_a_DataSet
    |-- a.py
    |-- x.py
    `-- x_data
        `-- array_0.npy

2 directories, 5 files
A small array
A list with a small and large array
[0 1 2 3 4 5 6 7 8 9]
[array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
↪ 10, 11, 12, 13, 14])]
```

As an alternative, you can directly import the dataset without the need for the `persist` library. This also has the feature of delayed loading:

[12]:
```
sys.path.append(tmpdir)
import dataset   # Only imports _info_dict at this point
print
print('import dataset: The dataset module initially contains')
print(dir(dataset))

import dataset.a, dataset.x   # Now we get a and x
print
print('import dataset.a, dataset.x: The dataset module now contains')
print(dir(dataset))
```

(continues on next page)

(continued from previous page)

```
sys.path.pop()

shutil.rmtree(tmpdir)        # Remove files
```

```
import dataset: The dataset module initially contains
['__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__
↪path__', '__spec__', '_info_dict']
import dataset.a, dataset.x: The dataset module now contains
['__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__
↪path__', '__spec__', '_info_dict', 'a', 'x']
```

CHAPTER 3

API

3.1 persist.archive

3.2 persist.interfaces

3.3 persist.objects

CHAPTER 4

Developer Notes

Here we check to see if we can use some of the code in the standard library `pickle.py` to simplify our life and solve issue #5.

[2]:
```
from io import StringIO
import pickle
file = StringIO()
p = pickle.Pickler(file)
```

4.1 Old-style classes

[4]:
```
# No special methods: __dict__ is copied
class A:
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
{'y': 2}
```

[5]:
```
# __setstate__ passed __dict__
# __dict__ not updated explicitly
class A:
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __setstate__(self, state):
        print('__setstate__({})'.format(state))
```

(continues on next page)

(continued from previous page)

```
a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
__setstate__({'y': 2})
{}
```

[6]:
```
# __getstate__ called instead of __dict__
class A:
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getstate__(self):
        print('__getstate__()')
        return dict(x=1)

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
__getstate__()
{'x': 1}
```

[7]:
```
# __getstate__ called and passed to __setstate__
class A:
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getstate__(self):
        print('__getstate__()')
        return dict(x=1)
    def __setstate__(self, state):
        print('__setstate__({})'.format(state))


a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
__getstate__()
__setstate__({'x': 1})
{}
```

[8]:
```
# __getstate__ called and passed to __setstate__
# Both __init__ called (without kw) and __setstate__ called
# __dict__ ignored
class A:
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getinitargs__(self):
        print('__getinitargs__()')
        return ('a', 3)
    def __getstate__(self):
```

(continues on next page)

(continued from previous page)

```
        print('__getstate__()')
        return dict(x=1)
    def __setstate__(self, state):
        print('__setstate__({})'.format(state))

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
__getstate__()
__setstate__({'x': 1})
{}
```

4.2 New-style classes

[9]:
```
# No special methods: __dict__ is copied
class A(object):
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a))
print(a1.__dict__)
```

```
__init__(*(), **{})
{'y': 2}
```

[10]:
```
# No special methods: __dict__ is copied
# __new__ is called for protocol >= 2
# __init__ is not called
class A(object):
    def __new__(cls, *v, **kw):
        print('__new__(*{}, **{})'.format(v, kw))
        return object.__new__(cls)
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a, protocol=2))
print(a1.__dict__)
```

```
__new__(*(), **{})
__init__(*(), **{})
__new__(*(), **{})
{'y': 2}
```

[11]:
```
# __getinitargs__ ignored in new-style classes
# __dict__ is still copied
class A(object):
    def __new__(cls, *v, **kw):
```

(continues on next page)

(continued from previous page)

```
        print('__new__(*{}, **{})'.format(v, kw))
        return object.__new__(cls)
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getinitargs__(self):
        print('__getinitargs__()')
        return ('a', 3)
a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a, protocol=2))
print(a1.__dict__)
```

```
__new__(*(), **{})
__init__(*(), **{})
__new__(*(), **{})
{'y': 2}
```

[12]:
```
# __getnewargs__ called for protocol >= 2
# __dict__ still copied
class A(object):
    def __new__(cls, *v, **kw):
        print('__new__(*{}, **{})'.format(v, kw))
        return object.__new__(cls)
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getnewargs__(self):
        print('__getnewargs__()')
        return ('a', 3)
a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a, protocol=2))
print(a1.__dict__)
```

```
__new__(*(), **{})
__init__(*(), **{})
__getnewargs__()
__new__(*('a', 3), **{})
{'y': 2}
```

[13]:
```
# __getnewargs__ called for protocol >= 2
# __dict__ still copied but from __getstate__ now
class A(object):
    def __new__(cls, *v, **kw):
        print('__new__(*{}, **{})'.format(v, kw))
        return object.__new__(cls)
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getnewargs__(self):
        print('__getnewargs__()')
        return ('a', 3)
    def __getstate__(self):
        print('__getstate__()')
        return dict(x=1)

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a, protocol=2))
```

(continues on next page)

(continued from previous page)

```
print(a1.__dict__)
```

```
__new__(*(), **{})
__init__(*(), **{})
__getnewargs__()
__getstate__()
__new__(*('a', 3), **{})
{'x': 1}
```

[14]:
```
# __getnewargs__ called for protocol >= 2
# __dict__ still copied but from __getstate__ now
class A(object):
    def __new__(cls, *v, **kw):
        print('__new__(*{}, **{})'.format(v, kw))
        return object.__new__(cls)
    def __init__(self, *v, **kw):
        print('__init__(*{}, **{})'.format(v, kw))
    def __getnewargs__(self):
        print('__getnewargs__()')
        return ('a', 3)
    def __getstate__(self):
        print('__getstate__()')
        return dict(x=1)
    def __setstate__(self, state):
        print('__setstate__({})'.format(state))

a = A()
a.y = 2
a1 = pickle.loads(pickle.dumps(a, protocol=2))
print(a1.__dict__)
```

```
__new__(*(), **{})
__init__(*(), **{})
__getnewargs__()
__getstate__()
__new__(*('a', 3), **{})
__setstate__({'x': 1})
{}
```

4.2.1 Summary

- With new-style classes, `__init__` is never called. One must define everything by updating `__dict__` or calling `__setstate__`.

[16]:
```
import sys
sys.path.insert(0, '../..')
import persist.archive
import persist.interfaces

a = persist.archive.Archive(scoped=False)
a.insert(f=persist.archive._from_pickle_state)
print(str(a))
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-fc296a285c9d> in <module>
```

(continues on next page)

(continued from previous page)

```
      5
      6 a = persist.archive.Archive(scoped=False)
----> 7 a.insert(f=persist.archive._from_pickle_state)
      8 print(str(a))

AttributeError: module 'persist.archive' has no attribute '_from_pickle_state'
```

[21]:
```
import numpy as np
import uncertainties
np.random.seed(3)
cov = np.random.random((3, 3))
x = [1, 2, 3]
u = uncertainties.correlated_values(
    nom_values=x, covariance_mat=cov, tags=['a', 'b', 'c'])
u = uncertainties.ufloat(0.1, 0.2)
```

[22]:
```
import persist.archive
a = persist.archive.Archive()
a.insert(u=u)
print(str(a))
```

```
Traceback (most recent call last):

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/IPython/core/
↪interactiveshell.py", line 3441, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-22-df2ce861ec8f>", line 4, in <module>
    print(str(a))

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 1265, in __str__
    res = self.scoped__str__()

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 1302, in scoped__str__
    graph = _Graph(objects=self.arch,

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 2311, in __init__
    node = self._new_node(obj, env, name)

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 2333, in _new_node
    rep, args, imports = self.get_persistent_rep(obj, env)

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 789, in get_persistent_rep
    return get_persistent_rep_repr(obj, env, rep=rep)

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 1644, in get_persistent_rep_repr
    _ast = AST(rep)

  File "/data/apps/conda/envs/work/lib/python3.8/site-packages/persist/archive.py",␣
↪line 2681, in __init__
```

(continues on next page)

(continued from previous page)

```
    self.__dict__['ast'] = ast.parse(expr)

  File "/data/apps/conda/envs/work/lib/python3.8/ast.py", line 47, in parse
    return compile(source, filename, mode, flags,

  File "<unknown>", line 1
    0.1+/-0.2
        ^
SyntaxError: invalid syntax
```

Table of Contents

4.3 Issue 11

[1]:
```
from persist.archive import Archive
x = [1, 2, 3]
y = [x, x]
a = Archive(scoped=False)
a.insert(y=y)
s = str(a)
d = {}
exec(s, d)
y_ = d['y']
assert y[0] is y[1]
assert y_[0] is y_[1]
```

After a bit of inspection, I found that the problem occurs at the reduction state. The call to `Graph.reduce()` is a bit too eager. Need to find out why. (Found a few bugs in the constructors for `Graph` and `Graph_` which should be similar...)

[3]:
```
from IPython.display import Image, display
import os
import sys
import trace

from persist.archive import Archive
x = [1, 2, 3]
y = [x, x]

from pycallgraph2 import PyCallGraph
from pycallgraph2.output import GraphvizOutput

images = []

a = Archive(scoped=True)
```

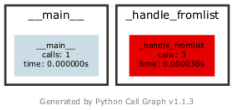
(continues on next page)

(continued from previous page)

```
a.insert(y=y)
with PyCallGraph(output=GraphvizOutput()):
    s = str(a)
images.append(Image(filename='pycallgraph.png', width="20%"))
os.remove('pycallgraph.png')

a = Archive(scoped=False)
a.insert(y=y)
with PyCallGraph(output=GraphvizOutput()):
    s = str(a)
images.append(Image(filename='pycallgraph.png', width="20%"))
os.remove('pycallgraph.png')

display(*images)
```



[4]:
```
g = a._graph
g.nodes
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-4249a64d2e6d> in <module>
----> 1 g = a._graph
      2 g.nodes

AttributeError: 'Archive' object has no attribute '_graph'
```

[5]:
```
# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=1,
    count=1)

# run the new command using the given tracer
tracer.run('str(a)')

# make a report, placing output in the current directory
r = tracer.results()
#r.write_results(show_missing=True, coverdir=".")
```

The question here is who is responsible for ensuring that objects are not duplicated? * The method `get_persistent_rep_list()` uses two names... so not here (but still has the same object).

4.4 Import Issues

The objective is to prevent the data from being loaded until `import ds.x` is called. This allows multiple processes to work with data independently with a lock file being required only when changing the metadata in `_info_dict`. To implement this we use the following trick suggested by Alex Martelli:

- Can modules have properties the same way that objects can?

Such a module might look like this:

```
import sys
import numpy as np
sys.modules[__name__] = np.array([0, 1, 2, 3])
```

or like this (if loaded from disk):

```
import os.path
import sys
import numpy as np
datafile = os.path.splitext(__file__)[0] + "_data.npy"
sys.modules[__name__] = np.load(datafile)
```

This seems to work very nicely. The imported array appears as part of the top-level module no matter how it is imported, and is only loaded when explicitly requested.

4.4.1 Byte Compiling

There is very subtle issue with using the import mechanism for DataSets. When updating an attribute of a DataSet, we change the corresponding `.py` file on disk. However, if this change is made too quickly after importing the attribute, it is possible that the byte-compiled `.pyc` file might not be finished compiling until *after* the `.py` file is updated. In this case, the `.py` file will have an earlier timestamp than the `.pyc` file and so python will incorrectly assume that the `.pyc` file is authoritative.

I do not see a good solution yet, so for now we use `` `sys.dont_write_bytecode <https://docs.python.org/2/library/sys.html?highlight=bytecode#sys.dont_write_bytecode>`__ ``:

- https://stackoverflow.com/a/154617

4.4.2 Reloading

Reloading data can also be an issue when replacing modules. For example, if we have a module `mod` and array `d` that is replaced, then what should `reload(mod)` do if `d` is updated on disk.

Our solution is the following:

- Delete all attributes like `mod.d` upon reload so that the user needs to re-import `mod.d` etc. This is done in the `__init__.py` file.
- This behaviour is inconstant with the normal python import machinery, so we only do it for objects that were specified with `single_item_mode`. The reason we do it here is that the user cannot `reload(mod.d)` since this is an array. If the user does not want this behaviour, then they can disable `single_item_mode`.
- The alternative behaviour might be to reload all data that has already been imported. We might provide a flag for this later if requested.

CHAPTER 5

Release Notes

As of version 3.1, we release only to PyPI using `poetry <https://python-poetry.org/>‘__. Here is the typical development/release cycle.

- First make sure you have a development environment with Mercurial, the evolve extension, topics enabled, [Black], [Nox], and [nbconvert].
- Set your virtual environment and run a shell to work in:

```
bash poetry env use python3.8 poetry shell poetry install -E doc -E test
```

- Start a development branch, i.e.:

```
hg branch 3.2
```

- Change version to `'3.2.dev0'` in `pyproject.toml` and commit this changes:

```
hg com -m "BRN: Start working on branch 3.2"
hg push --new-branch -r .
```

- Complete your changes making sure code is well tested etc. While working on specific features, you should always use topics:

```
hg topic new-feature
```

  When you push to Heptapod, the commits in these topics will remain in the draft phase, allowing you to rebase, etc. as needed to clean the history. We have setup automatic pushes to GitHub and you can see the status of the tests with the badge: .

  To run the tests locally, you should be able to just run:

```
nox
```

- Once everything is working and tested, push it to Heptapod and create Merge Requests:
- First merge all open topics to the development branch.

- Then change the revision in `pyproject.toml` to `'3.2'`, dropping the `'.dev'`. Push this to Heptapod and create a merge request to merge this to the default branch. Review the changes, and complete the Merge. Unlike previously, **do not close the branch.** Just leave it.
- Start work on next branch:

```
.. code:: bash
```

hg up 3.2 hg branch 3.3

5.1 PyPI

To release on PyPI:

‘‘‘bash poetry build poetry

```
hg up 3.2
poetry build

python setup.py sdist bdist_wheel
twine upload dist/persist-3.2*
```

5.2 Anaconda Cloud

To release on Anaconda Cloud (replace the filename as appropriate):

```
conda build meta.yaml
anaconda upload --all /data/apps/conda/conda-bld/osx-64/persist-3.0-py37_0.tar.bz2
```

CHAPTER 6

Indices and Tables

- genindex
- modindex
- search