
anamnesis Documentation

Release 1.0.0

Mark Hymers

Feb 27, 2020

Contents

1	Tutorials	3
1.1	Serialisation	3
1.2	MPI	13
2	Module reference	23
2.1	Serialisation	23
2.2	Class Registration	23
3	Introduction	25
4	Indices and tables	27

Contents:

1.1 Serialisation

1.1.1 Tutorial 1 - Using a Simple Serialised Object

The simplest use of anamnesis allows the serialisation of classes to and from hdf5 with relatively little extra code.

In order to both reading to and writing from HDF5 files to work, there are four basic steps

1. Inherit from the *anamnesis.AbstractAnam* class and call the class constructor
2. Ensure that your class constructor (*__init__*) can be called with no arguments (you may pass arguments to it but they must have default values)
3. Call the *anamnesis.register_class* function with the class
4. Populate the *hdf5_outputs* class variable with a list of member variable names necessary for serialisation/de-serialisation

Note that anamnesis uses the fully qualified class name when autoloading during the unserialising (loading) of object. If you want to use locally defined classes, you will have to ensure that they have been manually imported. For our examples, we will place our classes in the files *test_classesX.py* (where *X* is an ineger) and ensure that we can import this file into Python (i.e. it is on the *PYTHONPATH* or in the current working directory).

Our first example class is going to be a simple model of a person's name and age. We place the following code in *test_classes1.py*

```
#!/usr/bin/python3

from anamnesis import AbstractAnam, register_class

class SimplePerson(AbstractAnam):

    hdf5_outputs = ['name', 'age']
```

(continues on next page)


(continued from previous page)

```
def __init__(self, name='Unknown', age=0):
    AbstractAnam.__init__(self)

    self.name = name
    self.age = age

register_class(SimplePerson)
```

If we examine the *person* group in the HDF5 file, we can see that the class member variables:

File Window Help			
person			
	Name	Value	Type
	class	test_classes1.SimplePerson	<type 'str'> ()
	name	Fred	<type 'str'> ()
	age	42	int64 ()
HDF5 Attributes			

We can now write a script which will serial our data into an HDF5 file. We specify the group name when writing out.

```
#!/usr/bin/python3

import h5py

from test_classes1 import SimplePerson

# Create a person
s = SimplePerson('Fred', 42)

print(s.name)
print(s.age)

# Serialise the person to disk
f = h5py.File('test_script1.hdf5', 'w')
s.to_hdf5(f.create_group('person'))
f.close()
```

And write another script which loads the class back in. Because this class is not registered, we need to make sure that we have imported the module first. First of all, we can load from the file; if we know there is only one group in the file, we do not even need to specify the group name:

```
#!/usr/bin/python3
```

(continues on next page)

(continued from previous page)

```

from anamnesis import obj_from_hdf5file

import test_classes1 # noqa: F401

# Load the class from the HDF5 file
s = obj_from_hdf5file('test_script1.hdf5')

# Show that we have reconstructed the object
print(type(s))
print(s.name)
print(s.age)

# Demonstrate how to specifically choose which group to load
s2 = obj_from_hdf5file('test_script1.hdf5', 'person')

# Show that we have reconstructed the object
print(type(s))
print(s.name)
print(s.age)

```

If we want to load multiple objects from the same HDF5 file, we can open the file once and then use a function which loads from the group of the opened file. We can also tell anamnesis that it should autoload modules which start with a certain name. Both of these possibilities are demonstrated in the script `test_script1_read_B.py`:

```

#!/usr/bin/python3

import h5py

from anamnesis import obj_from_hdf5group, ClassRegister

# Register our class prefix so that we autoload our objects. This
# allows loading all classes whose fully resolved name
# starts with test_classes1; e.g. test_classes1.SimplePerson
ClassRegister().add_permitted_prefix('test_classes1')

# Open our HDF5 file
f = h5py.File('test_script1.hdf5', 'r')

# Load the class from the HDF5 file using our
# obj_from_hdf5group method
s = obj_from_hdf5group(f['person'])

# Show that we have reconstructed the object
print(type(s))
print(s.name)
print(s.age)

# Close our HDF5 file
f.close()

```

1.1.2 Tutorial 2 - More advanced serialisation features

Anamnesis has support for some more advanced features regarding serialisation.

In the main, most people will not require these, however they are used in NAF (the project from which anamnesis was

extracted).

These features can be best described by the name of the member variables or function names which are used to configure them.

One thing to note is that anamnesis implicitly reserves the use of these names for its own functionality. Note that any future additions will use the prefixes *hdf5_* or *anam_*. In order to avoid clashes with future versions of anamnesis, avoid using variables or function names with these prefixes.

1. *hdf5_defaultgroup* (member variable)
2. *hdf5_aliases* (member variable)
3. *hdf5_mapnames* (member variable)
4. *extra_data* (member variable)
5. *extra_bcast* (member variable)
6. *init_from_hdf5* (member function)
7. *refs* (member variable)
8. *shortdesc* (member variable)

The example classes used in this tutorial are placed in *test_classes2.py*.

```
#!/usr/bin/python3

from anamnesis import AbstractAnam, AnamCollection, register_class

class CollectableSubjectStats(AbstractAnam):

    hdf5_outputs = ['zstats', 'rts']

    hdf5_defaultgroup = 'subjectstats'

    def __init__(self, zstats=None, rts=None):
        """
        zstats must be a numpy array of [width, height] dimensions
        rts must be a numpy array of [ntrials, ] dimensions
        """
        AbstractAnam.__init__(self)

        self.zstats = zstats
        self.rts = rts

register_class(CollectableSubjectStats)

class StatsCollection(AnamCollection):
    anam_combine = ['zstats', 'rts']

register_class(StatsCollection)
```

All of the files needed to run these examples are generated by the script *test_script2_write.py*. This is also where several examples of the actual usage of the variables within classes can be seen.

```
#!/usr/bin/python3

import shutil

import h5py

from test_classes2 import (ComplexPerson,
                           ComplexPlace,
                           ComplexTrain)

# Create a person and a place
s = ComplexPerson('Anna', 45)

print(s.name)
print(s.age)

loc = ComplexPlace('York')
print(loc.location)

t = ComplexTrain('Glasgow')
print(t.destination)

# Serialise the person and place to disk
f = h5py.File('test_script2.hdf5', 'w')
s.to_hdf5(f.create_group(s.hdf5_defaultgroup))
loc.to_hdf5(f.create_group(loc.hdf5_defaultgroup))
t.to_hdf5(f.create_group(t.hdf5_defaultgroup))
f.close()

# Serialise the person to disk using a different name
# To do this, we copy the HDF5 file and manually edit it
shutil.copyfile('test_script2.hdf5', 'test_script2_aliases.hdf5')
f = h5py.File('test_script2_alias.hdf5', 'a')
f['person'].attrs['class'] = 'test_classes2.OldComplexPerson'
f.close()
```

hdf5_defaultgroup

This variable is usually used when serialising a single instance of a class into and out of an HDF5 file. Its use obviates the need to specify a group name when reading from an HDF5 file using the *from_hdf5file* function.

E.g., if we have two classes, one of which has an *hdf5_defaultgroup* set to *person* and the other to *place*, we can load each of the instances without specifying where they are in the file, as follows:

```
#!/usr/bin/python3

from test_classes2 import ComplexPerson, ComplexPlace # noqa: F401

# Load the classes from the HDF5 file using
# the default hdf5group names
s = ComplexPerson.from_hdf5file('test_script2.hdf5')
loc = ComplexPlace.from_hdf5file('test_script2.hdf5')

# Show that we have reconstructed the object
print("Person")
print(type(s))
```

(continues on next page)

(continued from previous page)

```
print(s.name)
print(s.age)

print("Place")
print(type(loc))
print(loc.location)
```

hdf5_aliases

hdf5_aliases is a list which allows developers to specify additional class names which should be matched by the given class. As an example, if *hdf5_aliases* in the *test_classes2.ComplexPerson* class is set to [*test_classes2.OldComplexPerson*], any files which were created using the old class name (*OldComplexPerson*) will now be read using the *ComplexPerson* class instead:

```
#!/usr/bin/python3

from anamnesis import obj_from_hdf5file

import test_classes2 # noqa: F401

# Demonstrate reading a file which has the old class name
# in the HDF5 file
s = obj_from_hdf5file('test_script2_aliases.hdf5', 'person')

# Show that we have reconstructed the object
print("Person")
print(type(s))
print(s.name)
print(s.age)
```

hdf5_mapnames

hdf5_mapnames is a rather specialised variable for which most users will not have a use. It allows users to control the mapping of variable names into and out of the HDF5 file - in other words, it decouples the names of the groups and attributes in the HDF5 file from those in the Python class.

As a concrete example, let us say that we are using a Python class which has a variable called *_order* but that for neatness sake, we would rather that this was called *order* in the HDF5 file. In this case, we would define the *hdf5_mapnames* variable as follows.

```
` hdf5_mapnames = {'_order': 'order'} hdf5_outputs = ['_order'] `
```

Note that *hdf5_mapnames* is a dictionary which maps Python class names to HDF5 entry names and that we still list the original variable name in *hdf5_outputs*.

You can have as many mappings as you want, but be very careful not to have a name in both *hdf5_outputs* and as a *target* in *hdf5_mapnames*. I.e., this is bad (assuming that your class has member variables *_order* and *myvariable*

```
` # Don't do this hdf5_mapnames = {'_order': 'myvariable'} hdf5_outputs =
['myvariable'] `
```

For (hopefully) obvious reasons, this makes no sense as you are attempting to serialise both the *_order* and *myvariable* variables into the HDF5 entry with name *myvariable*. Don't Do This (TM).

extra_data

The *extra_data* variable is a dictionary which can be used by users of a class to serialise and unserialise additional data which is not normally saved by the object.

To use this, simply use the *extra_data* as a standard dictionary, for example:

```
#!/usr/bin/python3

import h5py
from anamnesis import obj_from_hdf5file

from test_classes2 import ComplexPerson

# Create an example object
p = ComplexPerson('Bob', 75)
p.extra_data['hometown'] = 'Oxford'

print(p)
print(p.extra_data)

# Save the object out
f = h5py.File('test_script2_extradata.hdf5', 'w')
p.to_hdf5(f.create_group(p.hdf5_defaultgroup))
f.close()

# Delete our object
del p

# Re-load our object
p = obj_from_hdf5file('test_script2_extradata.hdf5')

# Show that we recovered the object and the extra data
print(p)
print(p.extra_data)
```

extra_bcast

The *extra_bcast* variable is a list of member variable names similar to that in the main *hdf5_outputs* variable. The difference is that variables listed in *extra_bcast* will be transferred via MPI when the object is sent or broadcast, but will *not* be placed into the HDF5 file during serialisation/unserialisation.

The most common use of this is when there is some cached information in the class which you do not want to recompute on every MPI node but do not need to save into the HDF5 file. In this case, the name of the variable containing the cache would *not* be listed in *hdf5_outputs* but would be listed in *extra_bcast*. It is also possible in that instance that you would wish to use the *init_from_hdf5* function as documented below.

init_from_hdf5

The optional function *init_from_hdf5* is called after the object has its members loaded when it is being unserialized from an HDF5 file. This means that you can perform any post-processing which you find necessary; for instance, if a class has a cache which needs updating after it is reinitialised (because it is not necessary to serialize/unserialize it), you can use this function to do so. To see how this works, look at the example class *ComplexTrain* in the *test_modules2.py* file shown above and examine the output from the *test_script2_initfromhdf5.py* script which uses this class:

```
#!/usr/bin/python3

from anamnesis import obj_from_hdf5file

from test_classes2 import ComplexPerson # noqa: F401

# Load the train object and watch for the printed output from the
# init_from_hdf5 function
p = obj_from_hdf5file('test_script2.hdf5', 'train')
```

refs

Full use of this variable requires the addition of anamnesis' report functionality. This will be ported from NAF soon.

shortdesc

Full use of this variable requires the addition of anamnesis' report functionality. This will be ported from NAF soon.

1.1.3 Tutorial 3 - AnamCollections - dealing with many similar objects

The *AnamCollection* code is designed to make it easy to deal with situations where you have many of the same type of class and want to perform analysis across them. For instance, in a sliding-window analysis, you may have a class which implements model fitting and metric measurement for a given window. You can then use an *AnamCollection* to easily collate the results together. You can consider *AnamCollection* to be a list of objects with additional helper functions to assist with the collation of data and HDF5 serialisation/deserialisation.

In general, you will want to subclass *AnamCollection* to use it. An example can be found in *test_classes3.py*:

```
#!/usr/bin/python3

from anamnesis import AbstractAnam, AnamCollection, register_class

class CollectableSubjectStats(AbstractAnam):

    hdf5_outputs = ['zstats', 'rts']

    hdf5_defaultgroup = 'subjectstats'

    def __init__(self, zstats=None, rts=None):
        """
        zstats must be a numpy array of [width, height] dimensions
        rts must be a numpy array of [ntrials, ] dimensions
        """
        AbstractAnam.__init__(self)

        self.zstats = zstats
        self.rts = rts

register_class(CollectableSubjectStats)
```

(continues on next page)

(continued from previous page)

```
class StatsCollection(AnamCollection):
    anam_combine = ['zstats', 'rts']

register_class(StatsCollection)
```

In this case, we have two variables, each of which will be a numpy array. The *AnamCollection* class is specifically designed for use with numpy arrays.

As an example of using this class when writing, you can see *test_script3_write.py*:

```
#!/usr/bin/python3

import h5py
import numpy as np

from test_classes3 import CollectableSubjectStats, StatsCollection

# Create a collection to put our data into
collection = StatsCollection()

# Simulate 5 peoples worth of data
for person in range(5):
    # 10x10 zstats - low resolution image!
    zstats = np.random.randn(10, 10)
    # 100 trials - averaging 450ms
    rts = np.random.randn(100) * 450.0

    p = CollectableSubjectStats(zstats, rts)

    collection.append(p)

# Write the data to a file
f = h5py.File('test_script3.hdf5', 'w')
collection.to_hdf5(f.create_group('data'))
f.close()
```

Note that objects can be appended into the collection object using the normal *.append()* method and then be written into an HDF5 file as normal.

When using a *AnamCollection* derived object, the simplest form of use is to treat it as a list which will let you retrieve the objects stored within it. This can be seen in the first few lines of the script below.

In addition, if you request any of the members which are referred to in the *anam_combine* member variable on the collection, the class will collate all of the instances of the identically named variable from the objects in the list and return an object which has these objects stacked. In most cases, you will use this with numpy arrays - you will then end up with a numpy array with an additional dimension. I.e., if each object has a numpy array of dimension (10, 10) and you have 3 objects, the combined array will have size (10, 10, 3). The objects are accessed by just accessing it as a member variable; for instance, if the name *data* was in *anam_combine* and your collection was named *collection*, you could access the combined data by accessing *collection.data*. Note that this member will only be available once you have called the *update_cache()* function on the collection - this is for reasons of efficiency. Therefore, after modifying, adding or deleting members in the list, you should call *update_cache()*. There is also a *clear_cache()* function but it is rarely used.

For a full example, see *test_script3_read.py*:

```
#!/usr/bin/python3

from anamnesis import obj_from_hdf5file

import test_classes3 # noqa: F401

# Load our collection of data
c = obj_from_hdf5file('test_script3.hdf5')

# Demonstrate how we have access to each individual object
for p in c.members:
    print(p.zstats.shape, p.zstats.min(), p.zstats.max())

# Make sure that our cache is up-to-date before we demonstrate
# the stacked data methods
c.update_cache()

# Demonstrate that we have access to stacked versions of the data
print(c.zstats.shape)
print(c.rts.shape)
```

1.1.4 Tutorial 4 - The *Store* class - saving data quickly

The *anamnesis.Store* class can be used as a very simple way of storing data without having to define your own class.

To use the *Store* class, you simply have to place any data which you want to serialise into the *extra_data* member variable.

As usual with the tutorials, we start with a script which creates an example test file: *test_script4_write.py*:

```
#!/usr/bin/python3

import h5py

from anamnesis import Store

# Create a Store
s = Store()

s.extra_data['airport_from'] = 'Manchester'
s.extra_data['airport_to'] = 'Schipol'

# Write the store into a file
f = h5py.File('test_script4.hdf5', 'w')
s.to_hdf5(f.create_group('data'))
f.close()
```

Reading back a *Store*

Reading back a *Store* is no different to reading any other anamnesis object as can be seen in *test_script4_read.py*.

```
#!/usr/bin/python3

from anamnesis import obj_from_hdf5file
```

(continues on next page)

(continued from previous page)

```
# Read from our store
s = obj_from_hdf5file('test_script4.hdf5')

print(s.extra_data['airport_from'])
print(s.extra_data['airport_to'])
```

1.2 MPI

1.2.1 Tutorial 1 - Broadcasting classes using MPI

As well as serialisation to and from HDF5, Anamnesis provides wrapper functionality to allow information to be sent between MPI nodes.

The use of the MPI functions in anamnesis requires the availability of the *mpi4py* module. If this is not available, you will not be able to use the MPI functions fully. You can, however, set *use_mpi=True* when creating the *MPIHandler()* object (see below) and then continue to use the functions. This allows you to write a single code base which will work both when doing multi-processing using MPI and running on a single machine.

The MPI functions require the same setup (primarily the *hdf5_outputs* class variable) as are used for the HDF5 serialisation / unserialisation, so we suggest that you work through the Serialisation tutorials first.

We are going to re-use some of the classes from the previous example. We place this code in *test_mpiclasses.py*

```
#!/usr/bin/python3

from anamnesis import AbstractAnam, register_class

class ComplexPerson(AbstractAnam):

    hdf5_outputs = ['name', 'age']

    hdf5_defaultgroup = 'person'

    def __init__(self, name='Unknown', age=0):
        AbstractAnam.__init__(self)

        self.name = name
        self.age = age

register_class(ComplexPerson)

class ComplexPlace(AbstractAnam):

    hdf5_outputs = ['location']

    hdf5_defaultgroup = 'place'

    def __init__(self, location='Somewhere'):
        AbstractAnam.__init__(self)

        self.location = location
```

(continues on next page)

(continued from previous page)

```

register_class(ComplexPlace)

class ComplexTrain(AbstractAnam):

    hdf5_outputs = ['destination']

    hdf5_defaultgroup = 'train'

    def __init__(self, destination='Edinburgh'):
        AbstractAnam.__init__(self)

        self.destination = destination

    def init_from_hdf5(self):
        print("ComplexTrain.init_from_hdf5")
        print("We have already set destination: {}".format(self.destination))

register_class(ComplexTrain)

```

We now write a simple Python script which uses the Anamnesis MPI interface. We will design this code so that the master node creates an instance of two of the classes and the slave nodes receive copies of these.

```

#!/usr/bin/python3

from anamnesis import MPIHandler

from test_mpiclasses import ComplexPerson, ComplexPlace, ComplexTrain

# All nodes must perform this
m = MPIHandler(use_mpi=True)

if m.rank == 0:
    # We are the master node
    print("Master node")

    # Create a person, place and train to broadcast
    s_person = ComplexPerson('Fred', 42)
    s_place = ComplexPlace('York')
    s_train = ComplexTrain('Disastersville')

    print("Master: Person: {} {}".format(s_person.name, s_person.age))
    print("Master: Place: {}".format(s_place.location))
    print("Master: Train to: {}".format(s_train.destination))

    m.bcast(s_person)
    m.bcast(s_place)
    m.bcast(s_train)

else:
    # We are a slave node
    print("Slave node {}".format(m.rank))

```

(continues on next page)

(continued from previous page)

```

# Wait for our objects to be ready
s_person = m.bcast()
s_place = m.bcast()
s_train = m.bcast()

print("Slave node {}: Person: {} {}".format(m.rank, s_person.name, s_person.age))
print("Slave node {}: Place: {}".format(m.rank, s_place.location))
print("Slave node {}: Train: {}".format(m.rank, s_train.destination))

# We need to make sure that we finalise MPI otherwise
# we will get an error on exit
m.done()

```

To run this code, we need to execute it in an MPI environment. As usual, make sure that anamnesis is on the *PYTHON-PATH*.

We can then call *mpirun* directly:

```

$ mpirun -np 2 python3 test_script1.py

Master node
Master: Person: Fred 42
Master: Place 1: York
Slave node 1
Master: Place 2: Glasgow
Slave node 1: Person: Fred 42
Slave node 1: Place 1: York
Slave node 1: Place 2: Glasgow

```

If you are using a cluster of some form (for instance *gridengine*), you will need to make sure that you have a queue with MPI enabled and that you submit your job to that queue. *Gridengine* in particular has good tight MPI integration which will transparently handle setting up the necessary hostlists.

The first thing which we need to do in the script is to initialise our *MPIHandler*. This is a singleton object and the *use_mpi* argument is only examined on the first use. This means that in future calls, you can call it without passing any argument.

```
m = MPIHandler(use_mpi=True)
```

In MPI, each instance of the script gets given a node number. By convention, we consider node 0 as the master node. All other nodes are designated as slave nodes. In order to decide whether we are the master node, we can therefore check whether our *rank* (stored on our *MPIHandler* object) is 0.

If we are the master, we then create three objects (a Person, a Place and a Train), set their attributes and print them out for reference. We then broadcast each of them in turn to our slave node or nodes.

On the slave node(s), we simply wait to receive the objects which are being sent from the master. There are two things to note. First, we do not need to specify the object type on the slave, this information is included in the MPI transfer. Second, we *must* make sure that our transmit and receive code is lined up; i.e. if we broadcast three items, every slave must receive three items. Code errors of this form are one of the most common MPI debugging problems. Try and keep your transmit / receive logic tidy and well understood in order to avoid long debugging sessions [#f1].

Once we have recieved the objects, we can simply use them as we normally would. Note that the objects are *not* shared before the two processes, you now have two distinct copies of each object.

Finally, it is important to call the *MPIHandler.done()* method to indicate to the MPI library that you have successfully finished.

Fotenotes

1.2.2 Tutorial 2 - Sending to/from different nodes

In many cases, we will not want to send data just from the master node to all other nodes. We can use a combination of *bcast*, *send* and *recv* to flexibly send around objects.

Again, for this example we are going to re-use some of the classes from the previous example which must be in *test_mpiclasses.py* (see Tutorial 1 for details).

Our new script looks like this:

```
#!/usr/bin/python3

import sys

from anamnesis import MPIHandler

from test_mpiclasses import ComplexPerson, ComplexPlace, ComplexTrain

# All nodes must perform this
m = MPIHandler(use_mpi=True)

# We need at least three nodes for this
if m.size < 3:
    print("Error: This example needs at least three MPI nodes")
    m.done()
    sys.exit(1)

if m.rank == 0:
    # We are the master node
    print("Master node")

    # Create a person to broadcast
    s_person = ComplexPerson('Fred', 42)

    print("Master: Created Person: {} {}".format(s_person.name, s_person.age))

    m.bcast(s_person)

    s_place = m.bcast(root=1)

    print("Master: Recieved Place: {}".format(s_place.location))

elif m.rank == 1:
    # We are slave node 1
    print("Slave node {}".format(m.rank))

    # Wait for our broadcast object to be ready
    s_person = m.bcast()

    print("Slave node {}: Recieved Person: {} {}".format(m.rank, s_person.name, s_
↪person.age))

    # Now create our own object and broadcast it to the other nodes
    s_place = ComplexPlace('Manchester')

    print("Slave node {}: Created place: {}".format(m.rank, s_place.location))
```

(continues on next page)

(continued from previous page)

```

m.bcast(s_place, root=1)

s_train = m.recv(source=2)

print("Slave node {}: Received Train: {} {}".format(m.rank, s_person.name, s_
↪person.age))
else:
    # We are slave node 2
    print("Slave node {}".format(m.rank))

    # Wait for our first broadcast object to be ready
    s_person = m.bcast()

    print("Slave node {}: Received Person: {} {}".format(m.rank, s_person.name, s_
↪person.age))

    # Wait for our second broadcast object to be ready
    s_place = m.bcast(root=1)

    print("Slave node {}: Received Place: {}".format(m.rank, s_place.location))

    # Create a train and send to node 1 only
    s_train = ComplexTrain('Durham')

    print("Slave node {}: Created train: {}".format(m.rank, s_train.destination))

    m.send(s_train, dest=1)

# We need to make sure that we finalise MPI otherwise
# we will get an error on exit
m.done()

```

Again, we need to run this code under an MPI environment (refer back to Tutorial 1 for details). We will get the following output:

```

Master node
Master: Created Person: Fred 42
Slave node 1
Slave node 2
Slave node 1: Recieved Person: Fred 42
Slave node 1: Created place: Manchester
Slave node 2: Received Person: Fred 42
Slave node 2: Received Place: Manchester
Master: Recieved Place: Manchester
Slave node 2: Created train: Durham
ComplexTrain.init_from_hdf5
We have already set destination: Durham
Slave node 1: Received Train: Fred 42

```

In order, our script does the following:

1. Set up MPI
2. Create a Person on node 0 (master) and *bcast* it to nodes 1 and 2
3. Create a Place on node 1 and *bcast* it to nodes 0 and 1

4. Create a Train on node 2 and *send* it to node 1 only (on which we call *recv*)

Using these examples, you should be able to see how we can flexibly send objects around our system.

1.2.3 Tutorial 3 - Scattering and Gathering data

Scattering and gathering numpy arrays

As well as broadcasting and transferring objects, we may wish to split data up for analysis. This is done using the *scatter_array* and *gather* functions.

In this script, we look at two ways of scattering data and then how to gather the data back up for consolidation:

```
#!/usr/bin/python3

import sys

import numpy as np

from anamnesis import MPIHandler

# All nodes must perform this
m = MPIHandler(use_mpi=True)

# We need at least three nodes for this
if m.size < 3:
    print("Error: This example needs at least three MPI nodes")
    m.done()
    sys.exit(1)

# Create a matrix of data for scattering
# Pretend that we have 300 points of data which we want to scatter,
# each of which is a vector of dimension 20

# This creates a matrix containing 0-19 in row 1,
# 100-119 in row 2, etc
data_dim = 20
num_pts = 300

if m.rank == 0:
    # We are the master node
    print("Master node")

    data = np.tile(np.arange((num_pts)) * 100, (data_dim, 1)).T + np.arange(data_
↪dim) [None, :]

    print("Master node: Full data array: ({} , {})".format(*data.shape))

    # 1. Scatter using scatter_array
    m1_data = m.scatter_array(data)

    print("Master node M1: m1_data shape: ({} , {})".format(*m1_data.shape))

    # 2. Scatter manually, using indices

    # Send the data to all nodes
    m.bcast(data)
```

(continues on next page)

(continued from previous page)

```

# Calculate which indices each node should work on and send them around
scatter_indices = m.get_scatter_indices(data.shape[0])
m.bcast(scatter_indices)

indices = range(*scatter_indices[m.rank])

m2_data = data[indices, :]

print("Master node M2: m2_data shape: ({} , {})".format(*m2_data.shape))

# 3. Gather using the gather function

# Create some fake data to gather
ret_data = (np.arange(m2_data.shape[0]) + m.rank * 100)[: , None]

print("Master node: data to gather shape: ({} , {})".format(*ret_data.shape))
print("Master node: first 10 elements: ", ret_data[0:10, 0])

all_ret_data = m.gather(ret_data)

print("Master node: gathered data shape: ({} , {})".format(*all_ret_data.shape))

print("all_ret_data 0:10: ", all_ret_data[0:10, 0])
print("all_ret_data 100:110: ", all_ret_data[100:110, 0])
print("all_ret_data 200:210: ", all_ret_data[200:210, 0])

else:
    # We are a slave node
    print("Slave node {}".format(m.rank))

    # 1. Scatter using scatter_array
    m1_data = m.scatter_array(None)

    print("Slave node {} M1: data shape: ({} , {})".format(m.rank, *m1_data.shape))

    # 2. Scatter manually, using indices
    # Recieve the full dataset
    data = m.bcast()

    # Get our indices
    scatter_indices = m.bcast()

    # Extract our data to work on
    indices = range(*scatter_indices[m.rank])

    m2_data = data[indices, :]

    print("Slave node {} M2: data shape: ({} , {})".format(m.rank, *m2_data.shape))

    # 3. Gather using the gather function

    # Create some fake data to gather
    ret_data = (np.arange(m2_data.shape[0]) + m.rank * 100)[: , None]

    print("Slave node {}: data to gather shape: ({} , {})".format(m.rank, *ret_data.
↪shape))

```

(continues on next page)

(continued from previous page)

```
print("Slave node {}: first 10 elements: ".format(m.rank), ret_data[0:10, 0])

m.gather(ret_data)

# We need to make sure that we finalise MPI otherwise
# we will get an error on exit
m.done()
```

Scattering Method 1: `scatter_array`

The simplest way to scatter data is to use the `scatter_array` function. This function always operates on the first dimension. I.e., if you have three nodes and a dataset of size $(100, 15, 23)$, the first node will receive data of size $(34, 15, 23)$ and the remaining two nodes $(33, 15, 23)$.

The code will automatically split the array unequally if necessary.

Scattering Method 2: `indices`

It is sometimes more useful to broadcast an entire dataset to all nodes using `bcast` and then have the nodes split the data up themselves (for instance, if they need all of the data for part of the computation but should only work on some of the data for the full computation).

To do this, we can use the `get_scatter_indices` function. This must be called with the size of the data which we are “scattering”. In the example in the text above, we would call this function with the argument `100`. The function then returns a list containing a set of arguments to pass to the `range` function. In the example above, this would be:

```
[(0, 34), (34, 67), (67, 100)]
```

There is an entry in the list for each MPI node. We broadcast this list to all MPI nodes which are then responsible for extracting just the part of the data required, for example (assuming that `m` is our `MPIHandler`):

```
all_indices = m.bcast(None)

my_indices = range(*all_indices[m.rank])
```

Note that these indices are congruent with the indices used during `gather`, so you can safely `gather` data which has been manually collated in this way.

Gathering numpy arrays

Gathering arrays is straightforward. Use the `gather` function, passing the partial array from each node. There is an example of this in `test_script3a.py` above. (Note that by default, the data is gathered to the root node).

Scattering and gathering lists

Scattering and gathering lists is similar to the process for arrays. There are two differences. The first is that you need to use the `scatter_list` and `gather_list` routines. The second is that the `gather_list` routine needs to be told the total length of the combined list, and on nodes where you want to receive the full list (including the master), you must pass `return_all` as `True` (the default is `False`).

An example script can be seen below:


```
#!/usr/bin/python3

import sys

from anamnesis import MPIHandler

# All nodes must perform this
m = MPIHandler(use_mpi=True)

# We need at least three nodes for this
if m.size < 3:
    print("Error: This example needs at least three MPI nodes")
    m.done()
    sys.exit(1)

# Create a list of data for scattering
# Pretend that we have 300 points of data which we want to scatter
num_pts = 300

if m.rank == 0:
    # We are the master node
    print("Master node")

    data = [str(x) for x in range(num_pts)]

    print("Master node: Full data array: len: {}".format(len(data)))

    # Scatter using scatter_list
    m1_data = m.scatter_list(data)

    print("Master node M1: m1_data len: {}".format(len(m1_data)))

    # Gather list back together again
    all_ret_data = m.gather_list(m1_data, num_pts, return_all=True)

    print("Master node: gathered list len: {}".format(len(all_ret_data)))

    print("all_ret_data 0:10: ", all_ret_data[0:10])
    print("all_ret_data 100:110: ", all_ret_data[100:110])
    print("all_ret_data 200:210: ", all_ret_data[200:210])
else:
    # We are a slave node
    print("Slave node {}".format(m.rank))

    # Scatter using scatter_list
    m1_data = m.scatter_list(None)

    print("Slave node {}: data len: {}".format(m.rank, len(m1_data)))

    # Gather using the gather_list function
    m.gather_list(m1_data, num_pts)

# We need to make sure that we finalise MPI otherwise
# we will get an error on exit
m.done()
```


CHAPTER 2

Module reference

2.1 Serialisation

2.2 Class Registration

CHAPTER 3

Introduction

anamnesis is a python module which provides the ability to easily serialise/unserialise Python classes to and from the HDF5 file format. It aims to be trivial to incorporate (normally requiring only a single extra class variable to be added to your classes) and flexible.

The library also extends the HDF5 serialisation/unserialisation capabilities to the MPI framework. This allows objects to be trivially passed between nodes in an MPI computation. The library also provides some wrapper routines to make it simpler to perform scatter and gather operations on arrays and lists (lists may even contain objects to be transferred).

anamnesis was originally written as part of the *NeuroImaging Analysis Framework*, a library intended for use in MEG theory work written at York NeuroImaging Centre, University of York, UK, but it has been split out in order to make it more generically useful.

CHAPTER 4

Indices and tables

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