HumPack

Release 0.3.1

Felix Leeb

Jan 03, 2021

INTRO

CHAPTER

ONE

INSTALL

Everything is tested with Python 3.7 on Ubuntu 18.04, but there is no reason it shouldn't also work for Windows.

You can install this package through pip:

```
pip install humpack
```

Alternatively, you can clone this repo and install the local version for development:

```
git clone https://github.com/felixludos/HumPack
pip install -e ./HumPack
```

CHAPTER

TWO

QUICKSTART

2.1 Containers

The provided containers: `tdict`, `tlist`, and `tset` serve as drop-in replacements for pythons `dict`, `list`, and `set` types that are `Transactionable` and `Packable` (more info below). Furthermore, all keys in `adict` that are valid attribute names, can be treated as attributes.

A few examples:

```
from humpack import adict, tdict, tlist, tset
from humpack import json_pack, json_unpack
from humpack import AbortTransaction

d = adict({'apple':1, 'orange':10, 'pear': 3})
d.apple += 10
d.update({'non-det banana':tset({2,3,7}), 'orange': None})
del d.pear
assert d.apple == 11 and 2 in d['non-det banana'] and 'pear' not in d
options = tlist(d.keys())
options.sort()
first = options[0]
assert first == 'apple'
d.order = options

json_d = json_pack(d)
assert isinstance(json_d, str)

d.begin() # starts a transaction (tracking all changes)
assert options.in_transaction()

d['non-det banana'].discard(7)
d.cherry = 4.2
assert 'cherry' in d and len(d['non-det banana']) == 2
d['order'].extend(['grape', 'lemon', 'apricot'])
assert 'grape' in options
del d.order[0]
del d['orange']
d.order.sort()
assert options[0] == 'apricot'

d.abort()
assert 'cherry' not in d and 7 in d['non-det banana']
assert 'grape' not in options
```

(continues on next page)

(continued from previous page)

```
with d:
    assert d['non-det banana'].in_transaction()
    d.clear()
    assert len(d) == 0
    d.melon = 100j
    assert 'melon' in d and d['melon'].real == 0
    raise AbortTransaction

assert 'melon' not in d

assert json_pack(d) == json_d
assert sum(d['non-det banana']) == sum(json_unpack(json_d)['non-det banana'])

with d:
    assert 'cherry' not in d
    d.cherry = 5
    # automatically commits transaction on exiting the context if no exception is␣
↪thrown

assert 'cherry' in d
```

When starting with data in standard python, it can be converted to using the "t" series counter parts using `containerify`.

```
from humpack import containerify
from humpack import AbortTransaction

x = {'one': 1, 1:2, None: ['hello', 123j, {1,3,4,5}]}

d = containerify(x)

assert len(x) == len(d)
assert len(x[None]) == len(d[None])
assert x['one'] == d.one
with d:
    assert d[None][-1].in_transaction()
    del d.one
    d.two = 2
    d[None][-1].add(1000)
    assert d['two'] == 2 and 'one' not in d and sum(d[None][-1]) > 1000
    raise AbortTransaction
assert 1000 not in d[None][-1] and 'one' in d and 'two' not in d
```

Finally, there are a few useful containers which don't have explicit types in standard python are also provided including heaps and stacks: `theap` and `tstack`.

2.2 Packing (serialization)

To serializing an object into a human-readable, json compliant format, this library implements packing and unpacking. When an object is packed, it can still be read (and manipulated, although that not recommended), converted to a valid json string, or encrypted/decrypted (see the Security section below). However for an obejct to be packable it and all of it's submembers (recursively) must either be primitives (`int`, `float`, `str`, `bool`, `None`) or registered as a `Packable`, which can be done

Packing and unpacking is primarily done using the `pack` and `unpack` functions, however, several higher level func-

tions are provided to combine packing and unpacking with other common features in object serialization. For custom classes to be `Packable`, they must implement three methods: `__pack__`, `__create__`, `__unpack__` (for more info see the documentation for `Packable`). When implementing these methods, all members of the objects that should be packed/unpacked, must use `pack_member` and `unpack_member` to avoid reference loops.

```
from humpack import pack, unpack

x = {'one': 1, 1:2, None: ['hello', 123j, {1,3,4,5}]}

p = pack(x) # several standard python types are already packable
assert isinstance(p, dict)
deepcopy_x = unpack(p)
assert repr(x) == repr(deepcopy_x)

from humpack import json_pack, json_unpack # Convert to/from json string

j = json_pack(x)
assert isinstance(j, str)
deepcopy_x = json_unpack(j)
assert repr(x) == repr(deepcopy_x)


from humpack import save_pack, load_pack # Save/load packed object to disk as json␣
↪file
import os, tempfile

fd, path = tempfile.mkstemp()
try:
    with open(path, 'w') as tmp:
        save_pack(x, tmp)
    with open(path, 'r') as tmp:
        deepcopy_x = load_pack(tmp)
finally:
    os.remove(path)
assert repr(x) == repr(deepcopy_x)
```

For examples of how to any types can registered to be `Packable` or objects can be wrapped in `Packable` wrappers, see the `humpack/common.py` and `humpack/wrappers.py` scripts.

2.3 Transactions

For examples of how `Transactionable` objects behave see the "Containers" section above.

To enable transactions for a class, it must be a subclass of `Transactionable` and implement the four required functions: `begin`, `in_transaction`, `commit`, and `abort`. Assuming these functions are implemented as specified (see documentation), you can manipulate instances of these classes in a transaction and then roll back all the changes by aborting the transaction.

One important thing to note with subclassing `Transactionable`: any members of instances of `Transactionable` subclasses should be checked for if they are also `Transactionable`, and if so, they the call should be delegated. In the example below, `Account` has to take into account that its attribute `user` could be `Transactionable`.

```
from humpack import Transactionable

class Account(Transactionable):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, user, balance=0.):
        super().__init__()
        self._in_transaction = False
        self._shadow_user = None

        self.user = user
        self.balance = balance

    def change(self, delta):

        if self.balance + delta < 0.:
            raise ValueError
        self.balance += delta

    def begin(self):
        # FIRST: begin the transaction in self
        self._shadow_user = self.user.copy(), self.balance # Assuming `user` can be␣
↪shallow copied with `copy()`
        self._in_transaction = True

        # THEN: begin transactions in any members that are Transactionable
        if isinstance(self.user, Transactionable):
            self.user.begin()

        # To be extra safe, you could also check `self.balance`, but we'll assume it
↪'s always a primitive (eg. float)

    def in_transaction(self):
        return self._in_transaction

    def commit(self):
        # FIRST: commit the transaction in self
        self._in_transaction = False
        self._shadow_user = None

        # THEN: commit transactions in any members that are Transactionable
        if isinstance(self.user, Transactionable):
            self.user.commit()

    def abort(self):
        # FIRST: abort the transaction in self
        if self.in_transaction(): # Note that this call only has an effect if self␣
↪was in a transaction.
            self.user, self.balance = self._shadow_user

        self._in_transaction = False
        self._shadow_user = None

        # THEN: abort transactions in any members that are Transactionable
        if isinstance(self.user, Transactionable):
            self.user.abort()
```

Optionally, for a more pythonic implementation, you can use `try`/`except` statements instead of type checking with `isinstance`.

2.4 Security

There are a few high-level cryptography routines. Nothing special, just meant to make integration in larger projects simple and smooth.

CHAPTER

THREE

PHILOSOPHY

[incomplete]

Focus is not on performance but on high-level features for fast and easy development.

3.1 Serialization

Object serialization tends to focus on storing all the information necessary for restoring the state of an object in the most compressed form possible. This is especially important when transmitting information across processes or a network, however there are also downsides, including: cross-platform compatibility and future-proofing. By using a human readable format, it's easier to implement an interpreter

CHAPTER

FOUR

PACKING

register_packable(*cls*, *pack_fn*, *create_fn*, *unpack_fn*, *name=None*)

Register a type to be packable. Requires a pack_fn, create_fn, and unpack_fn to store and restore object state.

Parameters

- **cls** (ClassVar) – type to be registered
- **pack_fn** (Callable) – callable input is an instance of the type, and packs all data necessary to recover the state
- **create_fn** (Callable) – callable input is the expected type and the packed data, creates a new instance of the type,

without unpacking any packed data (to avoid reference loops) :type unpack_fn: Callable :param unpack_fn: callable input is the instance of packed data and then restores that instance to the original state using the packed data by unpacking any values therein. :type name: Optional[str] :param name: (optional) name of the class used for storing :rtype: NoReturn :return: A *SavableClassCollisionError* if the name is already registered

class Packable

Bases: object

Any subclass of this mixin can be serialized using *pack*

All subclasses must implement __create__, __pack__, and __unpack__ to register the type. By passing a type to *use_cls* the type for which these methods are used can be overridden from the subclass.

classmethod __init_subclass__(*use_cls=None*, *name=None*)

This method automatically registers any subclass that is declared.

Parameters **use_cls** – The class to register (if it is different than *cls*)

Returns None

__deepcopy__(*memodict=None*)

Produces a deep copy of the data by packing and repacking.

Parameters **memodict** (Optional[Dict[Any, Any]]) – Unused

Return type Any

Returns A deep copy of self

classmethod __create__(*data*)

Create the object without loading the state from data. You can use the data to inform how to initialize the object, however no stored objects should be unpacked (to avoid reference loops)

Parameters **data** (Dict[str, NewType()(PACKED, object)]) – packed data to restore object state, should NOT be unpacked here

Return type *Packable*

Returns A fresh instance of the class registered with this create_fn

__pack__()
Collect all data in self necessary to store the state.

> **Warning:** All data must be "packed" storing it. This is done by passing the data into

Packable._pack_obj and using what is returned.

Return type Dict[str, NewType()(PACKED, object)]

Returns A dict of packed data necessary to recover the state of self

__unpack__(*data*)
Using *data*, recover the packed state. Must be overridden by all subclasses.

> **Warning:** All data must be "unpacked" before using it. This is done by passing the data into

Packable._unpack_obj and using what is returned.

Parameters **data** (Dict[str, NewType()(PACKED, object)]) – The information that is returned by *__pack__*.

Return type NoReturn

Returns Nothing. Once returned, the object should be in the same state as when it was packed

PRIMITIVE = typing.Union[str, int, float, bool, NoneType]
Valid primitives

SERIALIZABLE(*x*)

JSONABLE(*x*)

pack_member(*obj*, *force_str=False*)
Store the object state by packing it, possibly returning a reference.

This function should be called inside implemented __pack__ on all data in an object necessary to restore the object state.

Note: this function should not be called on the top level (use *pack* instead).

Parameters

- **obj** (NewType()(*SERIALIZABLE*, object)) – serializable data that should be packed
- **force_str** (bool) – if the data is a key for a dict, set this to true to ensure the key is a str

Return type NewType()(PACKED, object)

Returns packed data

unpack_member(*data*)
Restore the object data by unpacking it.

This function should be called inside implemented __unpack__ on all data in an object necessary to restore the object state from the packed data.

Note: this function should not be called on the top level (use *unpack* instead).

Parameters **data** (NewType()(PACKED, object)) – packed data that should be unpacked

Return type NewType()(*SERIALIZABLE*, object)

Returns unpacked data to restore the state

pack(*obj*, *meta=None*, *include_timestamp=False*)
Serializes any object, returning a json object that can be converted to a json string.

Parameters

- **obj** (NewType()(*SERIALIZABLE*, object)) – Object to be serialized
- **meta** (Optional[Dict[str, NewType()(PACKED, object)]]) – Meta information, must be jsonable
- **include_timestamp** (bool) – include a timestamp in the meta information

Return type NewType()(*JSONABLE*, object)

Returns packed data, which can be converted to a json string using json.dumps

unpack(*data*, *return_meta=False*)
Deserialize a packed object to recover the original state.

Parameters

- **data** (NewType()(PACKED, object)) – serialized (packed) state of an object
- **return_meta** (bool) – return any meta information from the serialized data

Return type NewType()(*SERIALIZABLE*, object)

Returns the unpacked (restored) object

save_pack(*obj*, *fp*, *meta=None*, *include_timestamp=False*)
Pack (serialize) the object and store it as a json file

Parameters

- **obj** (NewType()(*SERIALIZABLE*, object)) – object to be packed
- **fp** (TextIO) – writable file-like object where the packed object is stored
- **include_timestamp** (bool) – include timestamp in meta information

Return type NoReturn

Returns None

load_pack(*fp*, *return_meta=False*)
Loads json file of packed object and unpacks the object

Parameters

- **fp** (TextIO) – writable file-like object
- **return_meta** (bool) – return the meta information stored

Return type NewType()(*SERIALIZABLE*, object)

Returns unpacked object from json file

json_pack(*obj*, *meta=None*, *include_timestamp=False*)
Pack object and return a json string of the serialized object

Parameters

- **obj** (NewType()(*SERIALIZABLE*, object)) – to be packed
- **meta** (Optional[Dict[str, NewType()(*JSONABLE*, object)]]) – any meta information to include

- **include_timestamp** (bool) – include timestamp in meta information

Return type str

Returns json string of the serialized data

json_unpack(*data*, *return_meta=False*)

Unpack json string of a packed object.

Parameters

- **data** (str) – json string of a packed object
- **return_meta** (bool) – return meta information

Return type NewType()(*SERIALIZABLE*, object)

Returns unpacked object

CHAPTER

FIVE

TRANSACTIONABLE

class Transactionable

Bases: `object`

Mixin to enable beginning, committing, and aborting transactions (multiple statements). To use Transactionable functionality, subclasses must implement begin(), in_transaction(), commit(), and abort().

begin()

Must be overridden by subclasses. This prepares *self* to track all changes to *self* until commit() or abort() is called. If any attributes or data kept by *self*, this method should probably also call begin() in them.

Has no effect if *self* is already in a transaction.

Return type `NoReturn`

Returns None

in_transaction()

Query whether *self* is in a transaction.

Return type `bool`

Returns True if and only if *self* is currently in a transaction.

commit()

Make all the changes to *self* since transaction began, and stop tracking changes from now on.

Has no effect if *self* is not in a transaction.

Return type `NoReturn`

Returns None

abort()

Revert all changes to *self* since the transaction began, and stop tracking changes from now on.

Has no effect if *self* is not in a transaction.

Return type `NoReturn`

Returns None

__enter__()

Return type `NoReturn`

__exit__(*type*, **args*)

Once the context is ended, if no exception was raised the transaction is committed, otherwise,

Parameters

- **type** (Optional[Type]) – Either None or an Exception type, if an Exception was raised in the context.
- **args** (Any) – Other possible args provided by the raised Exception

Return type Optional[Type]

Returns If there was no exception, or the

exception AbortTransaction

Bases: Exception

CHAPTER

SIX

CONTAINERS

class Container

Bases: *humpack.transactions.Transactionable*, *humpack.packing.Packable*, *humpack.hashing.Hashable*

class tdict(**args*, ***kwargs*)

Bases: *humpack.basic_containers.Container*, collections.OrderedDict

Humpack dictionary, replaces the standard dict Has all the same functionality of a dict, plus being Transactionable or Packable

__init__(**args*, ***kwargs*)

in_transaction()

begin()

commit()

abort()

todict()

update(*other*)

fromkeys(*keys*, *value=None*)

clear()

copy()

__len__()

__hash__()

__eq__(*other*)

__contains__(*item*)

__reversed__()

__iter__()

keys()

values()

items()

pop(*key*)

popitem()

move_to_end(*key*, *last=True*)

__pack__()

__unpack__(*data*)

get(*k*, **args*, ***kwargs*)

setdefault(*key*, *default=None*)

__getitem__(*item*)

__setitem__(*key*, *value*)

__delitem__(*key*)

__str__(*default='{...}'*)

__repr__(*default='{...}'*)

class adict(**args*, ***kwargs*)
Bases: *humpack.basic_containers.tdict*

__getattr__(*item*)

__setattr__(*key*, *value*)

__delattr__(*item*)

class tlist(**args*, ***kwargs*)
Bases: *humpack.basic_containers.Container*, list

Humpack list, replaces the standard list Has all the same functionality of a list, plus being Transactionable or Packable

__init__(**args*, ***kwargs*)

in_transaction()

begin()

commit()

abort()

tolist()

copy()

__pack__()

__unpack__(*state*)

__getitem__(*item*)

__setitem__(*key*, *value*)

__delitem__(*idx*)

__hash__()

__eq__(*other*)

count(*object*)

append(*item*)

__contains__(*item*)

extend(*iterable*)

insert(*index*, *object*)

remove(*value*)

__iter__()

__reversed__()

reverse()

pop(*index=None*)

__len__()

clear()

sort(*key=None*, *reverse=False*)

index(*object*, *start=None*, *stop=None*)

__mul__(*other*)

__rmul__(*other*)

__add__(*other*)

__iadd__(*other*)

__imul__(*other*)

__str__(*default='[...]'*)

__repr__(*default='[...]'*)

class tset(*iterable=[]*)

Bases: *humpack.basic_containers.Container*, set

Humpack set, replaces the standard set Has all the same functionality of a set, plus being Transactionable or Packable

__init__(*iterable=[]*)

in_transaction()

begin()

commit()

abort()

toset()

copy()

__pack__()

__unpack__(*data*)

__hash__()

__eq__(*other*)

__and__(*other*)

__or__(*other*)

__xor__(*other*)

__sub__(*other*)

__rand__(*other*)

__ror__(*other*)

__rxor__(*other*)

__rsub__(*other*)

difference_update(*other*)

intersection_update(*other*)

union_update(*other*)

symmetric_difference_update(*other*)

symmetric_difference(*other*)

union(*other*)

intersection(*other*)

difference(*other*)

issubset(*other*)

issuperset(*other*)

isdisjoint(*other*)

__iand__(*other*)

__ior__(*other*)

__ixor__(*other*)

__isub__(*other*)

pop()

remove(*item*)

discard(*item*)

__contains__(*item*)

__len__()

__iter__()

clear()

update(*other*)

add(*item*)

__str__(*default='{...}'*)

__repr__(*default='{...}'*)

class tdeque(**args*, ***kwargs*)

Bases: *humpack.basic_containers.Container*, collections.deque

Humpack queue, replaces the standard deque Has all the same functionality of a set, plus being Transactionable or Packable

__init__(**args*, ***kwargs*)

in_transaction()

begin()

commit()

abort()

copy()

__pack__()

__unpack__(*state*)

__getitem__(*item*)

__setitem__(*key*, *value*)

__delitem__(*idx*)

__hash__()

__eq__(*other*)

count(*object*)

append(*item*)

appendleft(*item*)

__contains__(*item*)

extend(*iterable*)

extendleft(*iterable*)

insert(*index*, *object*)

remove(*value*)

__iter__()

__reversed__()

reverse()

pop()

popleft()

__len__()

clear()

sort(*key=None*, *reverse=False*)

index(*object*, *start=None*, *stop=None*)

rotate(*n=1*)

__mul__(*other*)

__rmul__(*other*)

__add__(*other*)

__iadd__(*other*)

__imul__(*other*)

__str__(*default='[...]'*)

__repr__(*default='[...]'*)

class tstack(**args*, ***kwargs*)

Bases: *humpack.basic_containers.tdeque*

Humpack stack Has all the same functionality of a deque, except it's a stack (FIFO) Also implements Transactionable and Packable

pop()

popend()

push(*item*)

push_all(*items*)

peek(*n=0*)

class _theap_iter(*heap*)
Bases: object

__init__(*heap*)

__next__()

class theap(**args*, ***kwargs*)
Bases: *humpack.basic_containers.Container*, object

Humpack heap Unordered for adding/removing, ordered when iterating. Note that iterating through the heap empties it.

__init__(**args*, ***kwargs*)

in_transaction()

begin()

commit()

abort()

copy()

__pack__()

__unpack__(*state*)

__iter__()

__len__()

push(**items*)

pop(*n=None*)

replace(*item*)

pushpop(*item*)

__hash__()

__eq__(*other*)

__str__(*default='[...]'*)

__repr__(*default='[...]'*)

containerify(*obj*, *dtype=<class 'humpack.basic_containers.tdict'>*)
Recursively, convert *obj* from using standard python containers to HumPack containers.

Parameters **obj** – object using python containers (dict, list, set, tuple, etc.)

Returns deep copy of the object using HumPack containers

CHAPTER

SEVEN

SECURE

format_key(*hsh*)
Reformat a hash (can be str or bytes)

Parameters **hsh** (Union[str, bytes]) – hash to be reformatted

Return type bytes

Returns a key which can be used for decrypting the data

secure_key(*word*, *salt=None*)
Get a hash from the raw text password

Parameters

- **word** (str) – raw text password
- **salt** (Optional[str]) – random salt to seed the hash computation

Return type str

Returns hash of the password

prompt_password_hash(*salt=None*)
Get the hash of a password which is entered by the user in a prompt

Parameters **salt** (Optional[str]) – random salt for hash computation

Return type str

Returns hash of the entered password

encrypt(*data*, *hsh=None*)
Encrypt the data from a provided hash (or prompt for a password if no hash is provided)

Parameters

- **data** (bytes) – bytes to be encrypted
- **hsh** (Union[str, bytes, None]) – hash used as key to encrypt

Return type bytes

Returns encrypted data bytes

decrypt(*data*, *hsh=None*)
Decrypt data with provided hash (or prompt for a password to compute hash if no hash is provided)

Parameters

- **data** (bytes) – encrypted bytes that should be decrypted

- **hsh** (Union[str, bytes, None]) – hash of a password to be used as a key to decrypt (obviously, must be the same as was used to encrypt)

Return type bytes

Returns the decrypted data, or a WrongKeyError if the key failed

secure_pack(*obj*, *hsh=None*, *meta=None*, *include_timestamp=False*)

Pack the object and encrypt it using the provided hash (prompt user for password if none is provided)

Parameters

- **obj** (NewType()(SERIALIZABLE, object)) – object to be packed
- **hsh** (Union[str, bytes, None]) – hash used as key to encrypt
- **meta** (Optional[Dict[str, NewType()(JSONABLE, object)]]) – meta information to store with the packed *obj*
- **include_timestamp** (bool) – include timestamp in meta info

Return type bytes

Returns encrypted bytes

secure_unpack(*data*, *hsh=None*, *return_meta=False*)

Decrypt *data* and unpack to recover the original object using the provided hash as a key

Parameters

- **data** (bytes) – encrypted bytes
- **hsh** (Union[str, bytes, None]) – hash to be used as a key to decrypt (prompt user, if not provided)
- **return_meta** (bool) – include meta info in output

Return type NewType()(SERIALIZABLE, object)

Returns decrypted and unpacked object, possibly including the meta info

CHAPTER

EIGHT

ERRORS

exception WrongKeyError
Bases: Exception

Error thrown when the provided key for decryption is incorrect

__init__()

exception LoadInitFailureError(*obj_type*)
Bases: Exception

Error thrown when an object cannot be recreated

__init__(*obj_type*)

exception ObjectIDReadOnlyError
Bases: Exception

Error thrown when trying to overwrite the obj_id

__init__()

exception SavableClassCollisionError(*addr*, *cls*)
Bases: Exception

Error thrown when trying to re-register a class already registered

__init__(*addr*, *cls*)

exception UnregisteredClassError(*name*)
Bases: Exception

Error thrown when trying to pack an instance of an unregistered class

__init__(*name*)

exception UnknownUserError
Bases: Exception

exception UnknownActionError
Bases: Exception

exception InsufficientPermissionsError(*user*, *action=None*)
Bases: Exception

__init__(*user*, *action=None*)

CHAPTER

NINE

HASHING

class Hashable

Bases: `object`

Mixin to allow hashing

__hash__()

__eq__(*other*)

CHAPTER

TEN

WRAPPERS

class Packable_Array

Bases: *humpack.packing.Packable*

Wrapper to allow saving numpy arrays. Aside from being rather useful, this serves as an example for how to write a Packable wrapper.

Note the necessary Packable methods are all static, and the use of "use_cls" in the class declaration.

static __create__(*data*)

Creates an empty np.array

Parameters **data** – packed data

Returns empty array with the correct size

static __pack__(*obj*)

Pack the np.array data.

Note: that the information necessary for creating thet instance (shape, dtype) is not packed, but still valid json objects

Parameters **obj** – instance of numpy.ndarray to be packed

Returns packed data

static __unpack__(*obj*, *data*)

Unpack the data and save the data to the created object

Parameters

- **obj** – instance with empty data to populate with the unpacked data
- **data** – packed data

Returns None

class ObjectWrapper(*obj*)

Bases: *humpack.transactions.Transactionable*, *humpack.packing.Packable*, ObjectProxy

Wrapper to transform an object to be transactionable.

Note: wrapped object must be copyable (shallow copy using *.copy()*)

WARNING: It is NOT recommended to use this wrapper, unless you need a transactionable features

__init__(*obj*)

begin()

in_transaction()

commit()

abort()

__repr__()

__str__()

__setattr__(*key*, *value*)

__delattr__(*item*)

__unpack__(*data*)

__pack__()
Save all the necessary data from the internal state (and pack any subdata)

Returns packed data

__build__(*data*)
Recover the wrapped object in the correct state from data and return wrapped object

Parameters **data** – packed data

Returns wrapped object with the loaded state

class Array(*obj*)
Bases: *humpack.wrappers.ObjectWrapper*

This is an example of how to use the *ObjectWrapper*. Wraps numpy arrays.

WARNING: it is NOT recommended to use this wrapper for numpy arrays (they are already registered).

begin()

commit()

abort()

__pack__()
Pack data to restore numpy array.

Returns packed data

__build__(*data=None*)
Restore state of numpy array by unpacking data

Parameters **data** – packed data

Returns restored state

CHAPTER

ELEVEN

COMMON

Here are a few commonly used non-standard python types that can be to made packable.

class Packable_Array

Bases: *humpack.packing.Packable*

Wrapper to allow saving numpy arrays. Aside from being rather useful, this serves as an example for how to write a Packable wrapper.

Note the necessary Packable methods are all static, and the use of "use_cls" in the class declaration.

static __create__(*data*)

Creates an empty np.array

Parameters **data** – packed data

Returns empty array with the correct size

static __pack__(*obj*)

Pack the np.array data.

Note: that the information necessary for creating thet instance (shape, dtype) is not packed, but still valid json objects

Parameters **obj** – instance of numpy.ndarray to be packed

Returns packed data

static __unpack__(*obj*, *data*)

Unpack the data and save the data to the created object

Parameters

- **obj** – instance with empty data to populate with the unpacked data
- **data** – packed data

Returns None

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

O

P

R

S

T

U

V

W