dill Documentation

Release 0.3.3.dev0

Mike McKerns

Aug 25, 2020

Contents:

CHAPTER 1

dill: serialize all of python

1.1 About Dill

`dill` extends python's `pickle` module for serializing and de-serializing python objects to the majority of the built-in python types. Serialization is the process of converting an object to a byte stream, and the inverse of which is converting a byte stream back to a python object hierarchy.

`dill` provides the user the same interface as the `pickle` module, and also includes some additional features. In addition to pickling python objects, `dill` provides the ability to save the state of an interpreter session in a single command. Hence, it would be feasable to save a interpreter session, close the interpreter, ship the pickled file to another computer, open a new interpreter, unpickle the session and thus continue from the 'saved' state of the original interpreter session.

`dill` can be used to store python objects to a file, but the primary usage is to send python objects across the network as a byte stream. `dill` is quite flexible, and allows arbitrary user defined classes and functions to be serialized. Thus `dill` is not intended to be secure against erroneously or maliciously constructed data. It is left to the user to decide whether the data they unpickle is from a trustworthy source.

`dill` is part of `pathos`, a python framework for heterogeneous computing. `dill` is in active development, so any user feedback, bug reports, comments, or suggestions are highly appreciated. A list of issues is located at https://github.com/uqfoundation/dill/issues, with a legacy list maintained at https://uqfoundation.github.io/pathos-issues.html.

1.2 Major Features

`dill` can pickle the following standard types:

- none, type, bool, int, long, float, complex, str, unicode,
- tuple, list, dict, file, buffer, builtin,
- both old and new style classes,
- instances of old and new style classes,
- set, frozenset, array, functions, exceptions

dill can also pickle more 'exotic' standard types:

- functions with yields, nested functions, lambdas,
- cell, method, unboundmethod, module, code, methodwrapper,
- dictproxy, methoddescriptor, getsetdescriptor, memberdescriptor,
- wrapperdescriptor, xrange, slice,
- notimplemented, ellipsis, quit

dill cannot yet pickle these standard types:

- frame, generator, traceback

dill also provides the capability to:

- save and load python interpreter sessions
- save and extract the source code from functions and classes
- interactively diagnose pickling errors

1.3 Current Release

This documentation is for version dill-0.3.3.dev0.

The latest released version of dill is available from:

https://pypi.org/project/dill

dill is distributed under a 3-clause BSD license.

```
>>> import dill
>>> dill.license()
```

1.4 Development Version

You can get the latest development version with all the shiny new features at:

https://github.com/uqfoundation

If you have a new contribution, please submit a pull request.

1.5 Installation

dill is packaged to install from source, so you must download the tarball, unzip, and run the installer:

```
[download]
$ tar -xvzf dill-0.3.2.tar.gz
$ cd dill-0.3.2
$ python setup py build
$ python setup py install
```

You will be warned of any missing dependencies and/or settings after you run the "build" step above.

Alternately, dill can be installed with pip or easy_install:

```
$ pip install dill
```

1.6 Requirements

`dill` requires:

- `python`, **version == 2.7** or **version >= 3.5**, or `pypy`

Optional requirements:

- `setuptools`, **version >= 0.6**
- `pyreadline`, **version >= 1.7.1** (on windows)
- `objgraph`, **version >= 1.7.2**

1.7 More Information

Probably the best way to get started is to look at the documentation at http://dill.rtfd.io. Also see `dill.tests` for a set of scripts that demonstrate how `dill` can serialize different python objects. You can run the test suite with `python -m dill.tests`. The contents of any pickle file can be examined with `undill`. As `dill` conforms to the `pickle` interface, the examples and documentation found at http://docs.python.org/library/pickle.html also apply to `dill` if one will `import dill as pickle`. The source code is also generally well documented, so further questions may be resolved by inspecting the code itself. Please feel free to submit a ticket on github, or ask a question on stackoverflow (**@Mike McKerns**). If you would like to share how you use `dill` in your work, please send an email (to **mmckerns at uqfoundation dot org**).

1.8 Citation

If you use `dill` to do research that leads to publication, we ask that you acknowledge use of `dill` by citing the following in your publication:

```
M.M. McKerns, L. Strand, T. Sullivan, A. Fang, M.A.G. Aivazis,
"Building a framework for predictive science", Proceedings of
the 10th Python in Science Conference, 2011;
http://arxiv.org/pdf/1202.1056

Michael McKerns and Michael Aivazis,
"pathos: a framework for heterogeneous computing", 2010- ;
https://uqfoundation.github.io/pathos.html
```

Please see https://uqfoundation.github.io/pathos.html or http://arxiv.org/pdf/1202.1056 for further information.

citation()
print citation

extend(*use_dill=True*)
add (or remove) dill types to/from the pickle registry

by default, `dill` populates its types to `pickle.Pickler.dispatch`. Thus, all `dill` types are available upon calling `'import pickle'`. To drop all `dill` types from the `pickle` dispatch, *use_dill=False*.

Parameters **use_dill** (*bool, default=True*) – if True, extend the dispatch table.

Returns None

license()
print license

load_types(*pickleable=True*, *unpickleable=True*)
load pickleable and/or unpickleable types to `dill.types`

`dill.types` is meant to mimic the `types` module, providing a registry of object types. By default, the module is empty (for import speed purposes). Use the `load_types` function to load selected object types to the `dill.types` module.

Parameters

- **pickleable** (*bool, default=True*) – if True, load pickleable types.
- **unpickleable** (*bool, default=True*) – if True, load unpickleable types.

Returns None

CHAPTER 2

dill module documentation

2.1 dill module

dill: a utility for serialization of python objects

Based on code written by Oren Tirosh and Armin Ronacher. Extended to a (near) full set of the builtin types (in types module), and coded to the pickle interface, by <mmckerns@caltech.edu>. Initial port to python3 by Jonathan Dobson, continued by mmckerns. Test against "all" python types (Std. Lib. CH 1-15 @ 2.7) by mmckerns. Test against CH16+ Std. Lib. ... TBD.

dump(*obj*, *file*, *protocol=None*, *byref=None*, *fmode=None*, *recurse=None*, ***kwds*)
pickle an object to a file

dumps(*obj*, *protocol=None*, *byref=None*, *fmode=None*, *recurse=None*, ***kwds*)
pickle an object to a string

load(*file*, *ignore=None*, ***kwds*)
unpickle an object from a file

loads(*str*, *ignore=None*, ***kwds*)
unpickle an object from a string

dump_session(*filename='/tmp/session.pkl'*, *main=None*, *byref=False*, ***kwds*)
pickle the current state of __main__ to a file

load_session(*filename='/tmp/session.pkl'*, *main=None*, ***kwds*)
update the __main__ module with the state from the session file

class Pickler(**args*, ***kwds*)
Bases: `pickle._Pickler`

python's Pickler extended to interpreter sessions

__init__(**args*, ***kwds*)
This takes a binary file for writing a pickle data stream.

The optional *protocol* argument tells the pickler to use the given protocol; supported protocols are 0, 1, 2, 3 and 4. The default protocol is 3; a backward-incompatible protocol designed for Python 3.

Specifying a negative protocol version selects the highest protocol version supported. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

The *file* argument must have a write() method that accepts a single bytes argument. It can thus be a file object opened for binary writing, an io.BytesIO instance, or any other custom object that meets this interface.

If *fix_imports* is True and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

__module__ = 'dill._dill'

_session = False

dispatch = {<class 'NoneType'>: <function _Pickler.save_none>, <class 'bool'>: <func

dump(*obj*)
Write a pickled representation of obj to the open file.

settings = {'byref': False, 'fmode': 0, 'ignore': False, 'protocol': 3, 'recurse':

class Unpickler(**args*, ***kwds*)
Bases: _pickle.Unpickler

python's Unpickler extended to interpreter sessions and more types

__dict__ = mappingproxy({'__module__': 'dill._dill', '__doc__': "python's Unpickler

__init__(**args*, ***kwds*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'dill._dill'

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

_session = False

find_class(*module*, *name*)
Return an object from a specified module.

If necessary, the module will be imported. Subclasses may override this method (e.g. to restrict unpickling of arbitrary classes and functions).

This method is called whenever a class or a function object is needed. Both arguments passed are str objects.

load()
Load a pickle.

Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

settings = {'byref': False, 'fmode': 0, 'ignore': False, 'protocol': 3, 'recurse':

register(*t*)
register type to Pickler's dispatch table

copy(*obj*, **args*, ***kwds*)
use pickling to 'copy' an object

pickle(*t*, *func*)
expose dispatch table for user-created extensions

pickles(*obj*, *exact=False*, *safe=False*, ***kwds*)
quick check if object pickles with dill

check(*obj*, **args*, ***kwds*)
check pickling of an object across another process

exception PicklingError
Bases: _pickle.PickleError

__module__ = '_pickle'

exception UnpicklingError
Bases: _pickle.PickleError

__module__ = '_pickle'

2.2 detect module

Methods for detecting objects leading to pickling failures.

baditems(*obj*, *exact=False*, *safe=False*)
get items in object that fail to pickle

badobjects(*obj*, *depth=0*, *exact=False*, *safe=False*)
get objects that fail to pickle

badtypes(*obj*, *depth=0*, *exact=False*, *safe=False*)
get types for objects that fail to pickle

code(*func*)
get the code object for the given function or method

NOTE: use dill.source.getsource(CODEOBJ) to get the source code

errors(*obj*, *depth=0*, *exact=False*, *safe=False*)
get errors for objects that fail to pickle

freevars(*func*)
get objects defined in enclosing code that are referred to by func

returns a dict of {name:object}

getmodule(*object*, *_filename=None*, *force=False*)
get the module of the object

globalvars(*func*, *recurse=True*, *builtin=False*)
get objects defined in global scope that are referred to by func

return a dict of {name:object}

nestedcode(*func*, *recurse=True*)
get the code objects for any nested functions (e.g. in a closure)

nestedglobals(*func*, *recurse=True*)
get the names of any globals found within func

outermost(*func*)
get outermost enclosing object (i.e. the outer function in a closure)

NOTE: this is the object-equivalent of getsource(func, enclosing=True)

referredglobals(*func*, *recurse=True*, *builtin=False*)
get the names of objects in the global scope referred to by func

referrednested(*func*, *recurse=True*)
get functions defined inside of func (e.g. inner functions in a closure)

NOTE: results may differ if the function has been executed or not. If len(nestedcode(func)) > len(referrednested(func)), try calling func(). If possible, python builds code objects, but delays building functions until func() is called.

trace(*boolean*)
print a trace through the stack when pickling; useful for debugging

varnames(*func*)
get names of variables defined by func

returns a tuple (local vars, local vars referrenced by nested functions)

2.3 objtypes module

all Python Standard Library object types (currently: CH 1-15 @ 2.7) and some other common object types (i.e. numpy.ndarray)

to load more objects and types, use dill.load_types()

2.4 pointers module

parent(*obj*, *objtype*, *ignore=()*)

```
>>> listiter = iter([4,5,6,7])
>>> obj = parent(listiter, list)
>>> obj == [4,5,6,7]  # actually 'is', but don't have handle any longer
True
```

NOTE: objtype can be a single type (e.g. int or list) or a tuple of types.

WARNING: if obj is a sequence (e.g. list), may produce unexpected results. Parent finds *one* parent (e.g. the last member of the sequence).

reference(*obj*)
get memory address of proxy's reference object

at(*address*, *module=None*)
get object located at the given memory address (inverse of id(obj))

parents(*obj*, *objtype*, *depth=1*, *ignore=()*)
Find the chain of referents for obj. Chain will end with obj.

objtype: an object type or tuple of types to search for depth: search depth (e.g. depth=2 is 'grandparents') ignore: an object or tuple of objects to ignore in the search

children(*obj*, *objtype*, *depth=1*, *ignore=()*)
Find the chain of referrers for obj. Chain will start with obj.

objtype: an object type or tuple of types to search for depth: search depth (e.g. depth=2 is 'grandchildren') ignore: an object or tuple of objects to ignore in the search

NOTE: a common thing to ignore is all globals, 'ignore=(globals(),)'

NOTE: repeated calls may yield different results, as python stores the last value in the special variable '_'; thus, it is often good to execute something to replace '_' (e.g. >>> 1+1).

2.5 settings module

global settings for Pickler

2.6 source module

Extensions to python's 'inspect' module, which can be used to retrieve information from live python objects. The methods defined in this module are augmented to facilitate access to source code of interactively defined functions and classes, as well as provide access to source code for objects defined in a file.

findsource(*object*)
Return the entire source file and starting line number for an object. For interactively-defined objects, the 'file' is the interpreter's history.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of all the lines in the file and the line number indexes a line in that list. An IOError is raised if the source code cannot be retrieved, while a TypeError is raised for objects where the source code is unavailable (e.g. builtins).

getsourcelines(*object*, *lstrip=False*, *enclosing=False*)
Return a list of source lines and starting line number for an object. Interactively-defined objects refer to lines in the interpreter's history.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An IOError is raised if the source code cannot be retrieved, while a TypeError is raised for objects where the source code is unavailable (e.g. builtins).

If lstrip=True, ensure there is no indentation in the first line of code. If enclosing=True, then also return any enclosing code.

getsource(*object*, *alias=''*, *lstrip=False*, *enclosing=False*, *force=False*, *builtin=False*)
Return the text of the source code for an object. The source code for interactively-defined objects are extracted from the interpreter's history.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An IOError is raised if the source code cannot be retrieved, while a TypeError is raised for objects where the source code is unavailable (e.g. builtins).

If alias is provided, then add a line of code that renames the object. If lstrip=True, ensure there is no indentation in the first line of code. If enclosing=True, then also return any enclosing code. If force=True, catch (TypeError,IOError) and try to use import hooks. If builtin=True, force an import for any builtins

indent(*code*, *spaces=4*)
indent a block of code with whitespace (default is 4 spaces)

outdent(*code*, *spaces=None*, *all=True*)
outdent a block of code (default is to strip all leading whitespace)

_wrap(*f*)
encapsulate a function and it's __import__

dumpsource(*object*, *alias=''*, *new=False*, *enclose=True*)
'dump to source', where the code includes a pickled object.

If new=True and object is a class instance, then create a new instance using the unpacked class source code. If enclose, then create the object inside a function enclosure (thus minimizing any global namespace pollution).

getname(*obj*, *force=False*, *fqn=False*)
get the name of the object. for lambdas, get the name of the pointer

_namespace(*obj); return namespace hierarchy (as a list of names*)
for the given object. For an instance, find the class hierarchy.

For example:

```
>>> from functools import partial
>>> p = partial(int, base=2)
>>> _namespace(p)
['functools', 'partial']
```

getimport(*obj*, *alias=''*, *verify=True*, *builtin=False*, *enclosing=False*)
get the likely import string for the given object

obj is the object to inspect If verify=True, then test the import string before returning it. If builtin=True, then force an import for builtins where possible. If enclosing=True, get the import for the outermost enclosing callable. If alias is provided, then rename the object on import.

_importable(*obj*, *alias=''*, *source=None*, *enclosing=False*, *force=True*, *builtin=True*, *lstrip=True*)
get an import string (or the source code) for the given object

This function will attempt to discover the name of the object, or the repr of the object, or the source code for the object. To attempt to force discovery of the source code, use source=True, to attempt to force the use of an import, use source=False; otherwise an import will be sought for objects not defined in __main__. The intent is to build a string that can be imported from a python file. obj is the object to inspect. If alias is provided, then rename the object with the given alias.

If source=True, use these options: If enclosing=True, then also return any enclosing code. If force=True, catch (TypeError,IOError) and try to use import hooks. If lstrip=True, ensure there is no indentation in the first line of code.

If source=False, use these options: If enclosing=True, get the import for the outermost enclosing callable. If force=True, then don't test the import string before returning it. If builtin=True, then force an import for builtins where possible.

importable(*obj*, *alias=''*, *source=None*, *builtin=True*)
get an importable string (i.e. source code or the import string) for the given object, including any required objects from the enclosing and global scope

This function will attempt to discover the name of the object, or the repr of the object, or the source code for the object. To attempt to force discovery of the source code, use source=True, to attempt to force the use of an import, use source=False; otherwise an import will be sought for objects not defined in __main__. The intent is to build a string that can be imported from a python file.

obj is the object to inspect. If alias is provided, then rename the object with the given alias. If builtin=True, then force an import for builtins where possible.

isdynamic(*obj*)
check if object was built in the interpreter

isfrommain(*obj*)
check if object was built in __main__

2.7 temp module

Methods for serialized objects (or source code) stored in temporary files and file-like objects.

dump_source(*object*, ***kwds*)

write object source to a NamedTemporaryFile (instead of dill.dump) Loads with "import" or "dill.temp.load_source". Returns the filehandle.

```
>>> f = lambda x: x**2
>>> pyfile = dill.temp.dump_source(f, alias='_f')
>>> _f = dill.temp.load_source(pyfile)
>>> _f(4)
16
```

```
>>> f = lambda x: x**2
>>> pyfile = dill.temp.dump_source(f, dir='.')
>>> modulename = os.path.basename(pyfile.name).split('.py')[0]
>>> exec('from %s import f as _f' % modulename)
>>> _f(4)
16
```

Optional kwds: If 'alias' is specified, the object will be renamed to the given string.

If 'prefix' is specified, the file name will begin with that prefix, otherwise a default prefix is used.

If 'dir' is specified, the file will be created in that directory, otherwise a default directory is used.

If 'text' is specified and true, the file is opened in text mode. Else (the default) the file is opened in binary mode. On some operating systems, this makes no difference.

NOTE: Keep the return value for as long as you want your file to exist !

dump(*object*, ***kwds*)

dill.dump of object to a NamedTemporaryFile. Loads with "dill.temp.load". Returns the filehandle.

```
>>> dumpfile = dill.temp.dump([1, 2, 3, 4, 5])
>>> dill.temp.load(dumpfile)
[1, 2, 3, 4, 5]
```

Optional kwds: If 'suffix' is specified, the file name will end with that suffix, otherwise there will be no suffix.

If 'prefix' is specified, the file name will begin with that prefix, otherwise a default prefix is used.

If 'dir' is specified, the file will be created in that directory, otherwise a default directory is used.

If 'text' is specified and true, the file is opened in text mode. Else (the default) the file is opened in binary mode. On some operating systems, this makes no difference.

NOTE: Keep the return value for as long as you want your file to exist !

dumpIO_source(*object*, ***kwds*)

write object source to a buffer (instead of dill.dump) Loads by with dill.temp.loadIO_source. Returns the buffer object.

```
>>> f = lambda x:x**2
>>> pyfile = dill.temp.dumpIO_source(f, alias='_f')
>>> _f = dill.temp.loadIO_source(pyfile)
```

(continues on next page)

(continued from previous page)

```
>>> _f(4)
16
```

Optional kwds: If 'alias' is specified, the object will be renamed to the given string.

dumpIO(*object*, ***kwds*)
dill.dump of object to a buffer. Loads with "dill.temp.loadIO". Returns the buffer object.

```
>>> dumpfile = dill.temp.dumpIO([1, 2, 3, 4, 5])
>>> dill.temp.loadIO(dumpfile)
[1, 2, 3, 4, 5]
```

load_source(*file*, ***kwds*)
load an object that was stored with dill.temp.dump_source

file: filehandle alias: string name of stored object mode: mode to open the file, one of: {'r', 'rb'}

```
>>> f = lambda x: x**2
>>> pyfile = dill.temp.dump_source(f, alias='_f')
>>> _f = dill.temp.load_source(pyfile)
>>> _f(4)
16
```

load(*file*, ***kwds*)
load an object that was stored with dill.temp.dump

file: filehandle mode: mode to open the file, one of: {'r', 'rb'}

```
>>> dumpfile = dill.temp.dump([1, 2, 3, 4, 5])
>>> dill.temp.load(dumpfile)
[1, 2, 3, 4, 5]
```

loadIO_source(*buffer*, ***kwds*)
load an object that was stored with dill.temp.dumpIO_source

buffer: buffer object alias: string name of stored object

```
>>> f = lambda x:x**2
>>> pyfile = dill.temp.dumpIO_source(f, alias='_f')
>>> _f = dill.temp.loadIO_source(pyfile)
>>> _f(4)
16
```

loadIO(*buffer*, ***kwds*)
load an object that was stored with dill.temp.dumpIO

buffer: buffer object

```
>>> dumpfile = dill.temp.dumpIO([1, 2, 3, 4, 5])
>>> dill.temp.loadIO(dumpfile)
[1, 2, 3, 4, 5]
```

capture(*stream='stdout'*)
builds a context that temporarily replaces the given stream name

```
>>> with capture('stdout') as out:
...   print "foo!"
...
>>> print out.getvalue()
foo!
```

CHAPTER 3

dill scripts documentation

3.1 get_objgraph script

display the reference paths for objects in `dill.types` or a .pkl file

Notes

the generated image is useful in showing the pointer references in objects that are or can be pickled. Any object in `dill.objects` listed in `dill.load_types(picklable=True, unpicklable=True)` works.

Examples:

```
$ get_objgraph FrameType
Image generated as FrameType.png
```

3.2 undill script

unpickle the contents of a pickled object file

Examples:

```
$ undill hello.pkl
['hello', 'world']
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

I

L

N

O

P

R

S

T

U

V