# pypersist Documentation

*Release 1.1*

**Michael Torpey**

**Apr 15, 2021**

# Contents:

pypi package 1.1

codecov 97%

docs passing

pypi package 1.1

launch binder

pypersist is a persistent memoisation framework for Python 3. Persistent memoisation is the practice of storing the output of a function permanently to a disk or a server so that the result can be looked up automatically in the future, avoiding any known results being recomputed unnecessarily.

# CHAPTER 1

# Installation

pypersist is available from PyPI, and the latest release can be installed using, for example:

```
pip3 install --user pypersist
```

Alternatively, the latest development version can be installed using Github:

```
git clone https://github.com/mtorpey/pypersist.git
pip3 install --user ./pypersist
```

CHAPTER 2

# Examples

To use, import the persist class from the pypersist package:

```
from pypersist import persist
```

and use it as a decorator when writing a function:

```
@persist
def double(x):
    return x * 2

print(double(3))
print(double(6.5))
```

This will store the outputs of the double function in a directory called persist/double/, in a machine-readable format.

One can specify various arguments to persist. For example:

```
@persist(key=lambda x,y: (x,y),
         hash=lambda k: '%s_to_the_power_of_%s' % k,
         pickle=str,
         unpickle=int)
def power(x, y):
    return x ** y

print(power(2,4))
print(power(10,5))
```

will store the outputs of power in human-readable files with descriptive filenames.

Many more options are available. See the persist class documentation for a full description, or launch the included notebook on Binder for more examples.

See this HackMD and the Issue tracker for current plans.

# CHAPTER 3

# Citing

Please cite this package as:

[Tor20] M. Torpey, pypersist, Python memoisation framework, Version X.Y (20XX), https://github.com/mtorpey/pypersist.

CHAPTER 4

# Acknowledgements

pypersist was created as part of the OpenDreamKit project: https://opendreamkit.org/

This part of the project is summarised in this report.

## 4.1 The `persist` decorator

The most important feature of pypersist is the `persist` decorator. As shown in the *Examples* section, you can use it by simply writing `@persist` above any function you wish to memoise.

`persist` can be used without any arguments, and its functionality will use use sane, conservative defaults. However, it can be customised in various ways using optional arguments, as follows.

`pypersist.`**`persist`** (*func=None*, *cache='file://persist/'*, *funcname=None*, *key=None*, *storekey=False*, *pickle=<function pickle>*, *unpickle=<function unpickle>*, *hash=<function hash>*, *unhash=None*, *metadata=None*, *verbosity=1*)

Function decorator for persistent memoisation

Store the output of a function permanently, and use previously stored results instead of recomputing them.

To use this, decorate the desired function with *@persist*. Or to customise the way this memoisation is done, decorate with *@persist(<args>)* and specify custom parameters.

You can even use this decorator for methods in a class. However, since it may be difficult to pickle a class instance, you may wish to specify a custom *key* function.

**Parameters**

- **`cache`** (`str, optional`) – The address of the cache in which the outputs of this function should be stored. If it starts with "file://", then the remainder of the string should be a path to the directory on the local file system in which the results will be stored; this may be a relative path, and the directory will be created if it does not exist. If it starts with "mongodb://" then the remainder of the string should be the URL of the pypersist MongoDB server in which the results will be stored. If it does not contain "://" then "file://" will be added at the beginning. Default is "file://persist".

- **funcname** (*str, optional*) – A string that uniquely describes this function. If the same *cache* is used for several memoised functions, they should all have different *funcname* values. Default is the name of the function.
- **key** (*function(args -> object), optional*) – Function that takes the arguments given to the memoised function, and returns a key that uniquely identifies those arguments. Two sets of arguments should have the same key only if they produce the same output when passed into the memoised function. Default returns a sorted tuple describing the arguments along with their names.
- **storekey** (*bool, optional*) – Whether to store the key along with the output when a result is stored. If True, the key will be checked when recalling a previously computed value, to check for hash collisions. If False, two keys will produce the same output whenever their *hash* values are the same. Default is False.
- **pickle** (*function(object -> str), optional*) – Function that converts the output of the function to a string for storage. Should be the inverse of *unpickle*. If *storekey* is true, this will also be used to store the key, and should do so without newline characters. Default uses the *pickle* module and base 64 encoding.
- **unpickle** (*function(str -> object), optional*) – Function that converts a string back to an object when retrieving a computed value from storage. Should be the inverse of *pickle*. If *storekey* is true, this will also be used to retrieve the key. Default uses the *pickle* module and base 64 encoding.
- **hash** (*function(object -> str), optional*) – Function that takes a key and produces a string that will be used to identify that key. If this function is not injective, then *storekey* can be set to True to check for hash collisions. The string should only contain characters safe for filenames. Default uses SHA-256 and base 64 encoding, which has an extremely small chance of collision.
- **unhash** (*function(str -> object), optional*) – Function that, if specified, should be the inverse of *hash*. If this is specified, it may be used whenever the keys of *cache* are requested. Default is None.
- **metadata** (*function( -> str), optional*) – Function that takes no arguments and returns a string containing metadata to be stored with the result currently being written. This might include the current time, or some data identifying the user or system that ran the computation.
- **verbosity** (*int, optional*) – What level of verbosity to output when running. If 0, nothing will be printed. If 1, prints only when something goes wrong. If 2, also prints when writing to files or clearing the cache. If 3, also prints when reading from the cache. If 4, it includes all the above with more details. Defaults to 1.

**Variables cache** (*diskcache.Cache or mongodb.Cache*) – Dictionary-like object that allows keys to be looked up and, if present, gives the previously computed value. Values can be added and removed using the syntax *func.cache[key] = val* and *del func.cache[key]*. If *storekey* is True or *unhash* is specified, this implements the collections.abc.MutableMapping abstract base class and we can iterate over its keys using *for key in func.cache*.

## Examples

Simple persistence using default settings:

```
>>> @persist
... def double(x):
```

(continues on next page)

(continued from previous page)

```
...     return 2 * x
>>> double(3)
6
>>> double(3)
6
>>> double.cache[(("x", 3),)]
6
```

Custom persistence using a simpler key, a descriptive filename, and writing human-readable files:

```
>>> @persist(key=lambda x,y: (x,y),
...          hash=lambda k: "%s_to_the_power_of_%s" % k,
...          pickle=str,
...          unpickle=int)
... def power(x, y):
...     return x ** y
>>> power(2,4)
16
>>> power(10,3)
1000
>>> power.cache[(2, 4)]
16
```

Persistence of a method inside a class. We specify a key function that characterises the relevant parts of the *A* object, since it can be difficult to pickle class instances:

```
>>> class A:
...     def __init__(self, x):
...         self.x = x
...     @persist(key=lambda self, a: (self.x, a))
...     def this_plus_number(self, a):
...         return self.x + a
>>> a = A(5)
>>> a.this_plus_number(10)
15
>>> a.this_plus_number.cache[(5, 10)]
15
>>> A.this_plus_number.cache[(5, 10)]
15
```

The default arguments used by `persist` rely on code in the following modules:

| Argument | Module |
|---|---|
| cache | *pypersist.diskcache* |
| key | *pypersist.preprocessing* |
| pickle, unpickle | *pypersist.pickling* |
| hash | *pypersist.hashing* |

## 4.2 Caching to a local disk

Persistent memoisation backend that saves results in the local file system

The *persist* decorator takes a *cache* argument, which details what sort of backend to use for the cache. If this string begins with "file://", or if no *cache* is specified, then a *disk cache* is used, which saves computed results to a directory in the local file system. This internal work is done by the classes defined below.

**class** `pypersist.diskcache.`**`Cache`**(*func*, *dir*)

Dictionary-like object for saving function outputs to disk

This cache, which can be used by the *persist* decorator in *persist.py*, stores computed values on disk in a specified directory so that they can be restored later using a key. Like a dictionary, a key-value pair can be added using *cache[key] = val*, looked up using *cache[key]*, and removed using *del cache[key]*. The number of values stored can be found using *len(cache)*.

A disk cache might not store its keys, and therefore we cannot iterate through its keys as we can with a dictionary. However, see *CacheWithKeys*.

**Parameters**

- **func** (*persist_wrapper*) – Memoised function whose results this is caching. Options which are not specific to local disk storage, such as the key, hash, and pickle functions, are taken from this.
- **dir** (*str*) – Directory into which to save results. The same directory can be used for several different functions, since a subdirectory will be created for each function based on its *funcname*.

**`clear`**()

Delete all the results stored in this cache

**class** `pypersist.diskcache.`**`CacheWithKeys`**(*func*, *dir*)

Mutable mapping for saving function outputs to disk

This subclass of *Cache* can be used in place of *Cache* whenever *storekey* is True or *unhash* is set, to implement the *MutableMapping* abstract base class. This allows the cache to be used exactly like a dictionary, including the ability to iterate through all keys in the cache.

**class KeysIter**(*cache*)

Iterator class for the keys of a *CacheWithKeys* object

**`next`**()

Return the next item from the iterator. When exhausted, raise StopIteration

## 4.3 Caching to a MongoDB database

Persistent memoisation backend that saves results on a MongoDB REST server

The *persist* decorator takes a *cache* argument, which details what sort of backend to use for the cache. If this string begins with “mongodb://”, then a *MongoDB cache* is used, which saves computed results to a MongoDB database via a REST API. This internal work is done by the classes defined below.

To start a MongoDB/REST server for use with this cache, navigate to the *mongodb_server/* directory and execute the *run.py* script.

**class** `pypersist.mongodbcache.`**`Cache`**(*func*, *url*)

Dictionary-like object for saving function outputs to disk

This cache, which can be used by the *persist* decorator in *persist.py*, stores computed values in a specified MongoDB database so that they can be restored later using a key. Like a dictionary, a key-value pair can be added using *cache[key] = val*, looked up using *cache[key]*, and removed using *del cache[key]*. The number of values stored can be found using *len(cache)*.

A MongoDB cache might not store its keys, and therefore we cannot iterate through its keys as we can with a dictionary. However, see *CacheWithKeys*.

**Parameters**

- **func** (*persist_wrapper*) – Memoised function whose results this is caching. Options which are not specific to local disk storage, such as the key, hash, and pickle functions, are taken from this.
- **url** (*str*) – URL of the pypersist MongoDB database that will be used to store and load results. The same database can be used for several different functions, since the function's *funcname* will be stored with each result.

**_get_db**(*hash=None*)

Return all db items for this function, or one with this hash

Queries the MongoDB database for entries with this function, and returns the resulting json data as a dictionary.

**Parameters** **hash** (*str, optional*) – The hash of the database item we wish to retrieve.

**Returns** If a hash is specified, a single database item with entries "_id", "_etag", "funcname", "hash", "result" and so on. If no hash is specified, a list of all such items in the database in the "_items" entry, along with metadata in the "_meta" entry. If no appropriate item exists in the database, None.

**Return type** dict or None

**clear**()

Delete all the results stored in this cache

**class** pypersist.mongodbcache.**CacheWithKeys**(*func, url*)

Mutable mapping for saving function outputs to a MongoDB database

This subclass of *Cache* can be used in place of *Cache* whenever *storekey* is True or *unhash* is defined, to implement the *MutableMapping* abstract base class. This allows the cache to be used exactly like a dictionary, including the ability to iterate through all keys in the cache.

**class KeysIter**(*cache*)

Iterator class for the keys of a *CacheWithKeys* object

**next**()

Return the next item from the iterator. When exhausted, raise StopIteration

## 4.4 Internal defaults

Several of the arguments to the `persist` decorator are functions. All of these have default values set in other modules in the package. Users interested in how these defaults behave can read the following manual sections for more information.

### 4.4.1 Processing arguments – `preprocessing`

Code to produce a key from a list of arguments to a function.

When a function decorated with *@persist* is called, its *key* function is called on the arguments to produce a key that corresponds to those arguments. If the user does not specify a custom *key* function, then we fall back to a default function that produces a tuple from the arguments; this tuple should be in a standard form that ignores functionally irrelevant features such as the ordering of keyword arguments. The *arg_tuple* function in this module produces this normalised tuple.

pypersist.preprocessing.**arg_tuple**(*func*, **args*, ***kwargs*)

Return a normalised tuple of arguments from args and kwargs

This function checks that *func(*args, **kwargs)* is a valid function call, then converts all the non-keyword arguments to keyword arguments. It then discards any supplied arguments that are equal to their default values (since they are unnecessary) and finally it sorts the arguments into alphabetical order by name. The arguments are then retuend as a tuple of tuples, where each tuple is a pair containing the name of the argument followed by the value given.

If *func* takes a variable number of arguments, any unnamed arguments will be included as a list with a name beginning with an asterisk.

**Parameters**

- **func** (*function*) – Function such that *func(*args, **kwargs)* is a valid call.
- **args** (*list*) – List of non-keyword arguments to be passed to *func*.
- **kwargs** (*dict*) – Dictionary of keyword arguments to be passed to *func*.

### Examples

Tuple of arguments to built-in function *len*:

```
>>> len("hello world")
11
>>> arg_tuple(len, "hello world")
(('obj', 'hello world'),)
```

Tuple of arguments to user-defined function, with default argument being discarded:

```
>>> def sum_of_three(x, a, m=2):
...     return x + a + m
>>> sum_of_three(10, m=2, a=15)
27
>>> arg_tuple(sum_of_three, 10, m=2, a=15)
(('a', 15), ('x', 10))
```

## 4.4.2 Pickling and unpickling objects – pickling

Default methods used by *persist* for pickling and unpickling objects.

pypersist.pickling.**pickle**(*obj*)

Return a string representation of the object *obj*

This function takes any object, and uses the *pickle* and *base64* modules to create a string which represents it. This string consists only of alphanumeric characters, hyphens and underscores. The object *obj* can later be reconstructed from this string using the *unpickle* function.

### Examples

```
>>> pickle("Hello world")
'gANYCwAAAEhlbGxvIHdvcmxkcQAu'
>>> unpickle("gANYCwAAAEhlbGxvIHdvcmxkcQAu")
'Hello world'
```

`pypersist.pickling.`**`pickle_to_bytes`**(*obj*)

Pickle an object to a bytes object

For most objects, this function is equivalent to *pickle.dumps*. However, if *pickle.dumps* fails, then an alternative pickling method will be attempted using Sage, if Sage is loaded. Otherwise, an error will be raised.

Used inside the *pickle* function in this file.

`pypersist.pickling.`**`unpickle`**(*string*)

Restore an object from a string created by the *pickle* function

If *string* was created by the *pickle* function in this file, then this function returns an object identical to the one that was used to create *string*.

#### Examples

```
>>> pickle("Hello world")
'gANYCwAAAEhlbGxvIHdvcmxkcQAu'
>>> unpickle("gANYCwAAAEhlbGxvIHdvcmxkcQAu")
'Hello world'
```

`pypersist.pickling.`**`unpickle_from_bytes`**(*obj*)

Unpickle a bytes object to produce the original object that was pickled

For most objects, this function is equivalent to *pickle.loads*. However, if *pickle.loads* fails, then an alternative unpickling method will be attempted using Sage, if Sage is loaded. Otherwise, an error will be raised.

Used inside the *unpickle* function in this file.

### 4.4.3 Hashing objects – `hashing`

Default method used by *persist* for hashing keys

`pypersist.hashing.`**`hash`**(*key*)

Return a string which is a hash of the argument given

It computes the SHA-256 sum of the key and returns it as a base 64 string. The string consists of alphanumeric characters, hyphens and underscores, and is precisely 43 characters long.

#### Examples

```
>>> hash("somestringkey123")
'wXS1bv_UbdX4riiyyA3Djjo7JeiEfyGI7o1-hGMnkz0'
>>> hash(3.141592654)
'nAh_dG9CDZL7bAFWX7E3iUXN2HXZ5eUiYUzdCJXDH-k'
>>> hash(None)
'Tz_DSKgYlBpGTkFf_2udQWwd3DscZHQ4YdMo-8NFvNY'
>>> key = (("arg1", [1,1,2,3,5,8,13]), ("x", "hello"))
>>> hash(key)
'1TBQNjqeAKCcCBmy-Sk_T1Xm01juuHOWiKotF5WYeZ8'
>>> hash("somestringkey123")
'wXS1bv_UbdX4riiyyA3Djjo7JeiEfyGI7o1-hGMnkz0'
```

# CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index