

# pyndb

- [Basics](#)
  - [About](#)
  - [Installation](#)
  - [Structure](#)
- [File Types](#)
  - [Naming Conventions](#)
  - [Pickling](#)
  - [JSON](#)
  - [Plaintext](#)
  - [Encryption](#)
- [Usage](#)
  - [Creating a new database](#)
  - [Saving](#)
  - [Creating new nodes](#)
  - [Retrieving values](#)
  - [Changing values](#)
  - [Finding values](#)
  - [Checking all values](#)
  - [Deleting Nodes](#)
  - [The transform method](#)
- [Additional Info](#)
  - [Known limitations](#)
  - [IDEs not playing nice](#)

- Tricks
  - Compression

# Basics

# About

This documentation covers version 3.4.3

## What is pyndb?

pyndb, short for Python Node Database, is a package which makes it easy to save data to a file while also providing syntactic convenience. It utilizes a Node structure which allows for easily retrieving nested objects. All data is wrapped inside of a custom Node object, and stored to file as nested dictionaries. It provides additional capabilities such as autosave, saving an existing dictionary to file, creating a file if none exists, and more. The original program was developed with the sole purpose of saving dictionaries to files, and was not released to the public.

## Why was it created?

pyndb was created when jvadair (the author) attempted to learn how to use sqlite with python. He decided that writing commands into strings and then passing them into a command execution function was too annoying. Thus DataManager was born. It was used on internal projects, until jvadair decided that that it too was quite annoying to use, due to it working poorly with nested dictionaries. Conceptualized on a sticky note late at night, he realized that he could recursively spawn objects within a master class. This solved the aforementioned issue, and so DataManager was rewritten into pyndb.

# Installation

## It's easy

To install, you can either install via pip:

```
pip install pyndb
```

Or, you can download the package folder and run:

```
python -m pip install dist/*.whl
```

# Structure

## How values are represented

All keys in a dictionary (loaded from file/object) managed by a PYNDatabase object will be represented as Node objects. Each Node object scans the dictionary it represents, and creates new Nodes for each key. This process repeats recursively. A Node object can also represent any other class, but you must then use the `transform` method - or replace the value with a dictionary - in order to create a subnode.

# File Types

# Naming Conventions

The proper (though not required) filename extension for a PYNDatabase is `.pyndb`

If saving to a JSON file, the prefix `.json` should be used instead.

As of version 3.2.0, pyndb will now automatically set the filetype for you based on the file's extension. The recognized extensions and their filetypes are as follows:

- `.pyndb` → pickled
- `.json` → json
- `.txt` or `.pydb` → plaintext (`.pydb` is included here for greater backwards compatibility)

All other files default to a pickled filetype.

“ Fun fact: In earlier versions not released to the public, the project was named DataManager and did not use the Node structure. Thus the proper filename extension was `.pydb`. This is partially how the name pyndb came to be.



# Pickling

pyndb saves data using `pickle` by default. `pickle` is installed with python, and thus should not require installation.

## Why pickle it?

Saving the data to file with `pickle` allows objects to be saved to file without force injecting a module into pyndb.

## Here's an example:

plaintext solution:

```
# Saving the class to file
import pyndb
import datetime
db = pyndb.PYNDatabase('example.pyndb', filetype='plaintext')
db.set('test', datetime.datetime.now())
db.save()

import pyndb
import datetime
pyndb.datetime = datetime
# This injection is necessary for plaintext, as the __repr__ value of the datetime object is what's saved to file.
# It also must be performed before you load the database.
db = pyndb.PYNDatabase('example.pyndb', filetype='plaintext')
print(db.test.val) # Will print the datetime object
```

pickled solution:

```
# Saving the class to file
import pyndb
import datetime
db = pyndb.PYNDatabase('example.pyndb')
db.set('test', datetime.datetime.now())
db.save()
```

```
import pyndb  
db = pyndb.PYNDatabase('example.pyndb')  
print(db.test.val) # Will print the datetime object
```

## How will this affect code from v2.x?

For now, backwards compatibility has been added, so when you attempt to open a plaintext file using the default setting (pickled) it will switch to plaintext for you. This will, however, throw a warning when the database loads, as this compatibility may be removed in the future. The option to set the file type to plaintext will remain, though.

## Converting plaintext to pickle

To convert a plaintext file to pickled data, simply set the `filetype` flag to `'pickled'` after initializing the `PYNDatabase`, and then call the `save` method.

## Using plaintext instead

The master class has a `filetype` flag, which can be changed like so:

```
# Change on initialization  
PYNDatabase(file, autosave=False, filetype='pickled')  
  
# Change after initialization  
PYNDatabase.filetype = 'plaintext'
```

# JSON

JSON files are useful when you want your data to be saved as simple, readable, and portable code. This makes it useful in things such as configuration files. JSON files, like plaintext, cannot store objects. To save a PYNDatabase to a JSON file, simply set the `filetype` flag to `json`.

# Plaintext

Plaintext format stores data exactly how you would expect it to - as plain text. What's actually written to the file is the string representation of the fileObj variable. Opening the file will reveal a single line of text being a python dictionary. This is useful for porting dictionaries across python scripts without pyndb, and also for large configurations that aren't meant to be read unless absolutely necessary. Like JSON data, it cannot store objects\*.

\*Objects can be stored if their `__str__` value is a command that recreates the object.

# Encryption

You can use encryption on any file type supported by pyndb.

You must have an up-to-date version of the cryptography module to use encryption features. Run "pip -U install cryptography" to update.

pyndb allows you to encrypt your databases using Fernet.

## Encrypting a new database

Simply specify a password when creating a database to encrypt it.

```
from pyndb import PYNDatabase
new_db = PYNDatabase('new_db.pyndb', password='your-password-here')
```

If you would like, you can also specify a salt and/or a number of iterations.

```
from pyndb import PYNDatabase
new_db = PYNDatabase('new_db.pyndb', password='a really long password', salt=b'A bytes object',
iterations=390000)
```

Due to limitations of PBKDF2, a salt and number of iterations must be provided. By default, pyndb uses `pyndb_default` as the salt, and 390000 as the number of iterations.

## Encrypting an existing database

if you load a database with the wrong password, it will try to load it without one. This is so that unencrypted databases can be loaded even if a password is specified. The password will also be set to `None` so that the database is not accidentally encrypted. Therefore, to encrypt an existing database, you can do as follows:

```
from pyndb import PYNDatabase
old_db = PYNDatabase('old_db.pyndb')
old_db.password = 'a very long password'
old_db.save()
```

# Removing encryption

To remove encryption, you can load the encrypted database with the password and then set it to None.

```
from pyndb import PYNDatabase
db = PYNDatabase('db.pyndb', password='your-password-here')
db.password = None
db.save()
```

# Usage

# Creating a new database

To create a new database, first import the PYNDatabase object from the module. Once you have done this, you can initialize it and then store it in a variable. A PYNDatabase object can be initialized with a filename (string), or a dictionary. You can also set the autosave flag, which will only work if a filename is set (otherwise the dictionary object will be updated automatically). These values CAN be changed later, by changing `PYNDatabase.<file | autosave>`.



# Saving

If a PYNDatabase object is initialized with a dictionary, it will update the original dictionary object as changes are made. Otherwise, you must call `PYNDatabase.save()` (unless autosave is set to `True`). The `save` method also has a `file` flag, which allows for easily saving to another file. The file type can be changed by setting the `filetype` parameter of the main class (see [Pickling](#)).

## Command usage:

```
PYNDatabase.save()
```

## Examples:

Basic usage:

```
from pyndb import PYNDatabase
db = PYNDatabase('filename.pyndb')
db.set('hello', 'world')
db.save() # <--
```

Saving an existing dictionary to file using the `file` flag

```
from pyndb import PYNDatabase
dictionary_obj = {'hello': 'world'}
db = PYNDatabase(dictionary_obj)
db.set('hello', 'world')
db.save(file='filename.pyndb') # <--
```

# Creating new nodes

Nodes can be created using the `set` or `create` methods.

The `set` method invokes `create` if the flag `create_if_not_exist` is set to `True`.

## Command Usage:

```
PYNDatabase.create(*names, val=None)
```

```
PYNDatabase.Node.create(*names, val=None)
```

Although you can create values using the `set` method, the `create` method will ultimately be called in order to do so. Additionally, the `create` method allows you to create multiple new Nodes.

If the `val` flag is set to `None` (default), then the new Nodes will have a `val` of `{}` (an empty dictionary). The reason that the `val` flag is set to `None` by default is due to the mutable default argument dilemma (see [076ad6b](#), [here](#)).

If ANY of the names specified already exists, NONE of them will be created.

## Examples:

1. Single/Multiple
2. What doesnt work (already exists (cancel whole operation))

```
from pyndb import PYNDatabase

db = PYNDatabse({}) # Creates a blank PYNDatabase from a new dict object
db.create('test') # Creates a single Node named test (Node.val = {})
db.create('test2', val='hello') # Creates a single Node named test2 with the value 'hello'
db.create('test3', 'testing', 'test4') # Creates multiple Nodes (Node.val = {})
db.create('test5', 'testing2', 'test6', val='hello') # Creates multiple Nodes with the value 'hello'

# This will not work!
db.create('test', 'testing3')

# Why?
# We already created <test> above.
```

# This means that NONE of the Nodes specified will be created.

# Retrieving values

Since values are stored as Node objects, object retrieval will look something like this:

```
PYNDatabase.Node.Node.val
```

`val` is a variable which contains the value of the Node, and is linked to the original dictionary object.

To dynamically retrieve a `Node`, you can use the `get` method. The `get` method is also the only way to retrieve values which contain characters not in the alphabet (+the underscore character).

This way, you can avoid writing code like this:

```
eval(f'PYNDatabase.Node.{name_of_node}.Node.val')
```

## Command Usage:

```
PYNDatabase.get(*names)
```

```
PYNDatabase.Node.get(*names)
```

If there is only 1 name given, the function will return a `Node` object. Otherwise, it will return a tuple of `Node`s.

## Examples:

### Static value retrieval

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')
db.set('hello', 'world')
print(db.hello.val) # <--
```

### Dynamically retrieving a value

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')
db.set('123', 456) # Note that the value type doesn't matter
```

```
# These will work...
print(db.get('123').val)

# OR
node_name = 'helloworld'
print(db.get(node_name).val)

# OR
node_names = ['helloworld', 'test']
print(db.get(*node_names)[0].val) # Displays the value of the first Node returned


# But these will not.
print(db.123.val)

# OR
node_names = ['helloworld', 'test']
print(db.get(*node_names).val) # Why doesn't this work?
# Because the object returned is a tuple, not a Node!
```

Storing a Node inside a variable is also a great option!

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')
db.set('hello', 'world')
hello = db.hello # <--
print(hello.val)
```

# Changing values

If a Node contains itself, a `RecursionError` will be thrown. See the [Structure](#) page for more info.

To change the value of a Node, you must use the `set` method from the parent Node.

`set` will create new values if they don't exist, but this can be changed by setting the `create_if_not_exist` flag to `False`. This way it will just raise a `NameError`.

## Command usage:

```
PYNDatabase.set(name, val, create_if_not_exist=True)
```

```
PYNDatabase.Node.set(name, val, create_if_not_exist=True)
```

## Examples:

Basic usage:

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')
db.set('hello', 'world') # <-- (creating value since none exists)
db.save() # Remember to save if needed!
```

Don't create new values:

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')

# This will work...
db.create('hello')
db.set('hello', 'world', create_if_not_exist=False)

# But this will not.
db.set('hello', 'world', create_if_not_exist=False)
```



# Finding values

To see if a Node with a specific name(s) is located within a parent, you can use the `has` method.

## Command usage:

```
PYNdatabase.has(*names)
```

```
PYNdatabase.Node.has(*names)
```

If multiple names are entered, True will be returned only if the Node has ALL the names.

## Examples:

Basic usage:

```
from pyndb import PYNDatabase
db = PYNDatabse('filename.pyndb')
db.set('hello', 'world')
db.set('helloagain', 'worldagain')
condition = db.has('hello') # <-- (will equal True)
condition = db.has('test') # <-- (will equal False)
condition = db.has('hello', 'helloagain') # <-- (will equal True)
condition = db.has('hello', 'test') # <-- (will equal False)
```



# Checking all values

This function will be replaced with a variable in the future if possible.

To view all the children inside a Node, you can call the `values` method (which is basically a glorified version of the `dir` command).

This command takes no arguments and returns a `list`.

Usage

# Deleting Nodes

To delete a Node, you can use the `delete` method.

## Command Usage:

```
PYNDatabase.delete(*names)
```

```
PYNDatabase.Node.delete(*names)
```

If ANY of the names specified do not exist, NONE of them will be deleted.

## Usage

# The transform method

The `transform` method places the existing value of a Node into a dictionary under a user-defined key. This way, you can create new Nodes inside your existing one.

Command usage:

```
PYNdatabase.transform(name, new_name)
```

```
PYNdatabase.Node.transform(name, new_name)
```

This method can be easily understood with the aid of a before and after diagram:

## Before

```
None
```

## After

```
{'new_name': None}
```

## Examples:

Basic usage:

```
from pyndb import PYNDatabase
db = PYNDatabase('filename.pyndb')
db.set('hello', 'world')
db.transform('hello', 'example') # <--
print(db.example.hello.val)
```

# Additional Info

# Known limitations

- Nodes can only be retrieved with the `get` method if they contain special characters or numbers (excluding `_`)
- Names such as `set`, `get`, `transform`, `val`, etc. (aka CoreNames) cannot be declared as they are already bound to methods/values that are required in order for the Node to function properly. There are also separate master-exclusive CoreNames such as `save`.
- There is no good way to change the master value (setting `PYNDatabase.fileObj` does not modify the Nodes within the database, only the master value.)
  - To work around this, simply redeclare the variable your PYNDatabase is stored in with a PYNDatabase object containing a dictionary, and set `PYNDatabase.file` to the filename (to overwrite).

# IDEs not playing nice

Some IDEs may throw errors saying that the PYNDatabase class does not have the attribute you requested. Technically, they're not wrong, as it hasn't been created yet. In PyCharm, the fix is simple: enter the context actions menu, and select `"Mark all unresolved attributes of PYNDatabase as ignored."` as shown in the image below:

[Pycharm fix](#)

Image not found or type unknown

# Tricks

Simple things that are not built-in but which could be useful

# Compression

Using `compress_pickle`, we can override `pyndb`'s default `save_pickle` and `load_pickle` functions (which are imported from `pickle`). Here's how it's done:

```
from compress_pickle import dump, load
import pyndb
pyndb.save_pickle = lambda obj, fn, *args: dump(obj, fn) # pyndb will try to send HIGHEST_PROTOCOL
pyndb.load_pickle = lambda fn, *args: load(fn)
```

Now, when opening a file, append the extension of the format you would like to use, for example:

```
db = pyndb.PYNDatabase("mnist.pyndb.gz", filetype="pickled")
```

Make sure to specify `filetype="pickled"` because `pyndb` does not recognize `.gz` by default.

To open the file, the same code will work.