

pyntree

- [Source code, roadmap, and more](#)
- [Getting started](#)
 - [What is pyntree?](#)
 - [A basic program with pyntree](#)
 - [Terminology](#)
 - [Installing](#)
- [Supported file types and actions](#)
 - [File types](#)
 - [The File object](#)
 - [Changing the active file](#)
 - [Reloading/retrieving the data from the file](#)
 - [Encrypting a file](#)
- [Basic Usage](#)
 - [Loading files/dictionaries](#)
 - [Getting nodes](#)
 - [Getting a value](#)
 - [Setting values](#)
 - [Saving](#)
 - [Getting the names of the children](#)
 - [Checking for a value](#)
 - [Deleting a Node](#)
 - [Getting the name of a Node](#)
 - [Math and object manipulation](#)
 - [Getting the children](#)

- Additional Features
 - The where() method
 - Converting back to a dictionary
 - Node representation
- Tips, Tricks, and Tidbits
 - Known limitations
 - Moving beyond a single flat file
 - How to fix pyntree installing as UNKNOWN-0.0.0
- Error messages
 - NameNotFound
 - FileNameUnset

Source code, roadmap, and more

Source code

pyntree's source code is available at <https://github.com/jvadair/pyntree>.

Roadmap

The roadmap is available at <https://board.jvadair.com>, but you will need to use the following credentials to access it:

- Username: public
- Password: public

I am looking for an alternative solution (and not Trello...)

Getting started

What is pyntree?

Overview

pyntree is a python package which allows you to easily and syntactically save your data. Not only that, it also lets you save in multiple formats, and even serialize and compress data by merely changing a few characters.

History

The first iteration of what is now known as pyntree came about when its sole developer decided that typing SQL commands into strings and running them was not a very pythonic way of doing things. Instead of dealing with this frustration, he created his own "database" system - DataManager. Although this system was never publicly released, pyndb, its successor, was (This is why pyndb was released on version 2 to start with). pyndb introduced a "node" system, in which each key of a dictionary was represented by a python object called a Node. pyntree keeps this structure, but its main difference is that nodes are created on the fly, via the `__getattr__` magic method.

Getting started

A basic program with pyntree

Let's take a simple, yet practical example.

Let's say you've written a web service, and need to save user data and be able to send it to clients quickly. You also need to load several config files.

Doing so with pyntree is easy and syntactic:

```
from pyntree import Node
from datetime import datetime

config = Node('config.json')
secrets_config = Node('secrets_config.json')
user_db = Node('users.pyn', autosave=True) # No need to call db.save() all the time!

# On event
user_db.get(user).data = new_data
user_db.get(user).data.time = datetime.now() # You can save objects directly to file!

# On user requesting data
send(user_db.get(user)()) # Calling the Node returns its value
```

Terminology

Node

A pythonic representation of data - each key in a dictionary is represented by a Node object. Node objects representing a dictionary "contain" (Nodes are created on-demand) other Nodes representing any keys in that dictionary.

Root node

This is your primary node - the one you initialize with file parameters and location (or a dictionary). The root node represents your main dictionary.

Child node

The root node spawns child nodes to represent all of the keys it contains. These child nodes can, again, spawn more nodes - but that does not make them root nodes.

Getting started

Installing

pyntree is available via pip:

```
pip install pyntree
```

That's it!

Supported file types and actions

Supported file types and actions

File types

pyntree currently supports a plethora of file types, all of which can be encrypted:

Pickle (default)

Pickled data is python data saved directly to a file as bytes. This way, you can save objects (ex: datetime) without losing any data or causing any problems while trying to reload the file.

Filetype parameter: `pyn`

Supported file extensions:

- .pyn
- .pyndb
- .pkl

Compressed (and pickled)

Compressed data can be saved in a variety of formats, thanks to the `compress_pickle` module. For a full list of supported file extensions and types, see [here](#).

JSON

JSON data can be saved and loaded with pyntree.

Filetype parameter: `json`

Supported file extensions:

- .json

YAML

YAML data can also be saved and loaded with pyntree.

Filetype parameter: `yaml`

Supported file extensions:

- .yaml

- .yaml

Plain text

Saving python objects to a plaintext file will likely cause loading issues since pyntree will use `eval()` to load the data. This means that if it tries to create an object from a class it hasn't imported, the loading will fail. Additionally, the reliance on the `eval()` function means that you should only load trusted plaintext files, as loading untrusted ones could present a security risk.

You can also save the data as a string representation of the dictionary.

Filetype parameter: `txt`

Supported file extensions:

- .txt

The File object

The File object is a helper object to ensure data consistency across nodes. Each node holds a link to the file. Although you don't have to use the File object directly, it is helpful to understand what parameters are available since they can be passed to your root node.

Parameters:

- data (Mandatory)
 - The filename or dictionary to be loaded into the File object
- filetype (Optional - auto-detected, fallback is 'pyn')
 - See [File types](#).
- autosave (Optional - False)
 - Save file on modification
- save_on_close (Optional - False)
 - Save data to the file when the File object is destroyed

Changing the active file

To change the active file, you can use the `switch_to_file` method:

```
File.switch_to_file(filename, filetype=None)  
# You can also call Node.switch_to_file for initialized Nodes.
```

If the `filetype` parameter is set to `None`, then the filetype will be inferred.

This method **will not** reload the data from the file.

To reload the data from the new file, see [Reloading/retrieving the data from the file](#).

Supported file types and actions

Reloading/retrieving the data from the file

To reload the data from the file, use the following method:

```
File.reload()
```

This method is only 1 line of code:

```
self.data = self.read_data()
```

The `read_data` method returns the data stored in the file, after being parsed into a python dictionary.

Encrypting a file

Encryption is a new feature as of 1.0.0!

To use encryption, first type `pip install pyntree[encryption]` to install the necessary dependencies.

To encrypt your data, simply specify a password parameter for the Node when saving or loading.

```
db = Node('encrypted.pyn', password='password')
```

Optionally, you can also specify a salt:

```
db = Node('encrypted.pyn', password='password', salt=b'random_salt')
```

Basic Usage

Loading files/dictionaries

Syntax:

```
Node(file_or_dictionary, **file_parameters)
```

To load a file/dictionary, simply create a new Node object with the filename or dictionary as first argument, along with any optional file arguments you want to pass (See [The File object](#)). For default filetype mappings, see [File types](#).

```
db = Node('file.pyn')
```

```
db = Node('file.abc', filetype='txt', autosave=True)
```

Getting nodes

Syntax:

```
Node.get(*children)
```

There are two ways to retrieve a child Node:

```
root_node.get('child')  
root_node.child
```

You can also retrieve deeper nodes like so:

```
root_node.a.b.c  
root_node.get('a').b.get('c')
```

Notice that it doesn't particularly matter whether you use the `.get()` method or simply access the Node as an attribute.

Remember, getting a node means accessing the child node object - to get the value, see [Getting values](#).

Getting a value

Syntax:

```
Node()
```

pyntree takes a somewhat unique yet simple approach to getting the values stored in nodes thanks to python limitations. To get a node's value, simply call it:

```
root_node()
```

The same applies to child nodes.

To get the names of all the child nodes, see [Getting a list of all children](#).

Setting values

Syntax:

```
Node.set(*children, value)
```

To change a value or create a new node, you can use one of two general methods:

The `set()` method:

```
your_node.set('name', 'Jimmy')
```

Or by directly setting the attribute:

```
your_node.name = 'Jimmy'
```

Remember, if a node doesn't exist to begin with, you can't create children of it!
The below code will NOT WORK:

```
your_node.doesnt_exist.value = 3
```

Saving

Syntax:

```
Node.save(filename=None)
```

Simply call the save method to save your data:

```
root_node.save()
```

You can also call the save method on child nodes, but it will save the data in the root node to the proper file.

You can also specify a different filename to (temporarily) save to:

```
root_node.save(filename="temp.file")
```

For changing the file to be saved to (in a more permanent sense), see [\[replaceme\]](#)

Getting the names of the children

Syntax:

```
Node._values
```

To get a list of all of a Node's children, simply use the `values` property:

```
Node._values
```

This will return a list of strings, not Nodes.

If you have a child Node named `"_values"`, you will need to use the `get` function to retrieve it. See [here](#) for more details.

`.values -> ._values` as of 1.0.0

Checking for a value

Syntax:

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

To see if a Node has a child with a given name, use the `has` method:

```
Node.has(name)
```

Deleting a Node

Syntax:

```
Node.delete(*children)
```

The delete function can take either 0 or 1+ parameters:

```
Node.delete()
```

```
Node.delete(name)
```

In the first example, the node on which the function is called will be deleted. In the latter, the child with the specified name will be deleted.

When we speak of "deleting nodes" here, we are referring to deleting the data which those Nodes represent.

Getting the name of a Node

Syntax:

```
Node._name
```

Let's say you have a miscellaneous Node that got passed to a function somehow. What is its purpose? The name of the Node may shed some light on this:

```
from pyntree import Node
x = Node('test.pyn')
x._name # 'test.pyn'
x.a = 1
x.a._name # 'a'
```

A root node without a filename (pure dictionary) will return the string 'None' (not the object, for compatibility reasons).

If you have a child Node named "_name", you will need to use the `get` function to retrieve it. See [here](#) for more details.

.name -> ._name as of 1.0.0

Math and object manipulation

Syntax:

```
Node += val
```

Math operations

If you used pyndb, you probably had to do something like this:

```
x = PYNDatabase({})  
x.set("a", 1)  
x.set("a", x.a.val + 1)
```

Let's be real here, this **SUCKS**. pyntree does it better:

```
x = Node()  
x.a = 1  
x.a += 1
```

Wow, that's infinitely simpler and less painful, right?!

Object operations

You can interact with a Node's data directly once you retrieve it:

```
x = Node()  
x.a = [1,2,3,4]  
x.a().append(5)  
print(x.a()) # -> [1,2,3,4,5]
```


Getting the children

Syntax:

```
Node._children
```

The `._children` property returns a list of Node objects (the children of the parent Node)

Additional Features

The where() method

Syntax:

```
Node.where(**kwargs)
```

Let's say you have a lot of users, sorted by IDs. Now, you want to find a user with a specific, unique email. Sounds like a pain, right? Nope. Here's how to do it with pyntree:

```
db = Node("users.pyn")  
my_user = db.where(email="their@email.com") # -> [Node({"id": ...})]
```

Keep in mind that this will always return a list of Nodes (or an empty list).

Thus, finding multiple users with the same name is no problem:

```
db = Node("users.pyn")  
my_user = db.where(name="John") # -> [Node({"id": ...}), Node({"id": ...}), ...]
```

Converting back to a dictionary

Syntax:

```
dict(Node)
```

This is pretty straightforward, as shown above. The command will make a new dictionary. If you want to directly manipulate a Node's data instead, you can simply perform methods on the called function, as shown in the section [Math and object manipulation](#).

Node representation

Syntax:

```
str(Node)
# or
repr(Node)
```

-
- `str(Node)` will return a string representation of the data stored within the Node
 - `repr(Node)` will return the same as `str(Node)`, but wrapped within the text "`Node()`"

Tips, Tricks, and Tidbits

Known limitations

- Attempting to retrieve a name which is also a Node property will not work as intended (ex: `_values`) when doing so explicitly (`Node._values`), but using the `get` method will yield the desired result. For example, if you have a data point/key with a sub-attribute of `_values`, running `Node._values` will work as [described here](#) rather than returning that sub-attribute. However, running `Node.get('_values')` will work as intended. In code, this looks like:

```
x = Node()
x._values = 1
x._values
Output: ['values']
x.get('_values')
Output: <pyntree.Node object at memory_location>
```

Moving beyond a single flat file

For more complex projects, a flat-file database may not be the way to go. Here is one example of how you could store data in a more distributed way:

- db folder
 - users
 - user1.pyn
 - user2.pyn
 - etc
 - transactions
 - 001.pyn
 - 002.pyn
 - etc
 - config.json

Python implementation:

```
from pyntree import Node
config = Node('db/config.json')

# On request with argument "transaction_id"
data = Node(f'db/transactions/{transaction_id}')
return data()
```

How to fix pyntree installing as UNKNOWN-0.0.0

```
pip install --upgrade pip wheel setuptools
```

Error messages

NameNotFound

This error means that there is no Node with the name that you provided.

Here's an example:

```
x = Node()
x.get('test')
# or
x.test
```

Output: `<RootNode>.test does not exist`

The error message will always write the text `<RootNode>` since the method returning will not be able to determine what you have named the variable containing the Node object. For more information, see [Terminology](#).

FileNameUnset

This error means that you tried to save a Node, but initialized it with raw data and never set a filename. **To fix this**, see [Changing the active file](#), or set the filename parameter on the save function to save temporarily (see [Saving](#)).

Here's an example:

```
x = Node()
x.get('test')
# or
x.test
```

Output:

You have not specified a filename for this data.

Try setting the filename parameter or use `switch_to_file`.