JSON Config Documentation

Release 0.3

Brian Peterson

Jul 17, 2018

Contents:

CHAPTER 1

Welcome to JSON Config

Configuration doesn't get any easier than this ...

1.1 Installation

```
pip install jsonconfig-tool
```

1.2 What's it Used For?

- Managing settings, configuration information, application data, etc.
- Managing secrets, tokens, keys, passwords, etc.
- Managing environment settings.

1.3 Basic Example

```
with Config('myapp') as cfg:
    cfg.data = 'Any JSON serializable object ...'
    cfg.pwd.a_secret = 'Encrypted data ...'
    cfg.env.a_variable = 'Environment variables.'
```

In the context manager above:

- The `data` is stored in the user's local *application directory*.
- The `pwd` data is encrypted and stored in a *keyring vault*.

- The `env` data is stored in *environment variables*.

1.4 Easy Access

With Python-Box[3] as a data access wrapper you can reference Mapping data as either dictionary-keys or attribute-style notation. For example,

```
with BoxConfig('myapp') as cfg:
    cfg.data.key1 = 'Some configuration data ...'
    cfg.data.key2 = 'Some more data ...'
```

1.5 Designed with Stability in Mind

- JSON Config is just a pass-through to mature, stable 3rd party packages, and built-in Python modules: Click[1], Keyring[2], Python-Box[3], open, json.load, json.dump, and os.environ.
- JSON Config takes extra care to stay out of your way as a Python programmer, it does NOT introduce magic. And any keyword arguments that you would normally be able to pass to the above functions are still available through the context manager.
- You could import each of the above packages independently rather than using JSON Config but then you'd be responsible for writing a lot more code, tracking what all needs to be imported, writing a lot more tests, and dealing with error handling from different sources. The more custom code you write, the greater the chance of introducing bugs.

1.6 Consistency across Multiple Sources of Configuration Data

- JSON Config aims to create consistency across different types of configuration data. For example, bringing both dictionary key and attribute-style access to data, encrypted data, and environment variables.
- JSON Config provides consistent error handling around the entire configuration process; this allows you to employ sane exception management and logging at the granularity-level most suitable to your project.
- JSON Config also simplifies the process of cross-checking configuration settings across multiple sources. An example might be first checking to see if an environment variable is set. If not set, then checking to see if the setting has been recorded in the configuration file, and if not prompting the user for the required setting and then writing it back to the configuration. In JSON Config this can all be done in one simple line of code.

1.6.1 References

[3] http://github.com/cdgriffith/Box
[1] http://github.com/pallets/click
[2] https://github.com/jaraco/keyring

CHAPTER 2

JSON Config Examples

2.1 Storing & Retrieving Database Server Settings

```
from jsonconfig import Config

# Save the settings.

with Conf('Mongo DB') as mongo:
  mongo.data = {"domain": "www.example.com",
    "mongodb": {"host": "localhost", "port": 27017}}

# Retrieve the settings.

with Conf('Mongo DB') as mongo:
    data = mongo.data
```

Retrieve the hostname & port from environments variables if they exist, otherwise pull the information from the configuration file if it exists, otherwise use default values.

```
from jsonconfig import Config

with Conf('Mongo DB') as db:
    host = db.env['M_HOST'] or db.data['mongodb']['host'] or 'localhost'
    port = int(db.env['M_PORT'] or db.data['mongodb']['port'] or 27017)
```

2.2 Storing & Retrieving the Current State of a Game

```
from jsonconfig import BoxConfig, Config

with Config('Chess Tournaments') as chess_match:
    chess_match.data = [
```

(continues on next page)

(continued from previous page)

```
        {
            "Event": "Kramnik - Leko World Championship Match",
            "Site": "Brissago SUI",
            "Date": "2004.10.03",
            "EventDate": None,
            "Round": 6,
            "Result": None,
            "White": {"Name": "Vladimir Kramnik", "Elo": None},
            "Black": {"Name": "Peter Leko", "Elo": None},
            "ECO": "C88",
            "PlyCount": 40,
            "Moves": [
                'e4 e5', 'Nf3 Nc6', 'Bb5 a6', 'Ba4 Nf6', 'O-O Be7',
                'Re1 b5', 'Bb3 O-O', 'h3 Bb7', 'd3 d6', 'a3 Na5'
            ]
        }
    ]

# Pick the game up where it left off and add new moves.

with BoxConfig('Chess Tournaments') as chess_matches:
    match = chess_matches[0].data
    match.Moves.append([
        'Ba2 c5', 'Nbd2 Nc6', 'c3 Qd7', 'Nf1 d5', 'Bg5 dxe4',
        'dxe4 c4', 'Ne3 Rfd8', 'Nf5 Qe6', 'Qe2 Bf8', 'Bb1 h6'
    ])
    match.Result = '1/2-1/2'

# Retrieve the data for all matches.

with Config('Chess Tournaments') as chess_matches:
      match = chess_matches.data
```

2.3 Making a Request from a Restful API Settings File

```
from jsonconfig import Config

api_info = {
    "headers": {"Authorization: Bearer {access_token}"}
    "parameters": {"includeAll": True}
    "resources": {"sheet": {"endpoint": "sheets/{sheetId}"}}
)

# example of saving both standard and enrypted data

with Config('Sample API') as api:

    # save standard data
    api.data = api_info

    # save encyrpted data
    api.pwd.access_token = 'll352u9jujauoqz4gstvsae05'

# example of updating an existing configuration
```

(continues on next page)

(continued from previous page)

```
with Config('Sample API') as api:
    api.data['url'] = "https://api.smartsheet.com/2.0/"
```

Pull access token from environment variable if it exists, otherwise pull it from the Keyring vault, it it doesn't exist there either prompt the user for the password and mask the characters as they're typed in.

```
import requests
from jsonconfig import getpass, BoxConfig

with BoxConfig('API Example') as api:
    endpoint = api.data.resources.sheet.endpoint
    access_token = api.env.ACCESS_TOKEN or api.pwd.access_token or getpass
    response = requests.get(
        api.data.url + endpoint.format(sheetId=4583173393803140),
        headers = api.data.headers.format(access_token),
        parameters = api.data.parameters
    )
```

CHAPTER 3

Configuration File Basics

3.1 Definition

A configuration file is used for the persistent storage of data, like when a user shuts down a program, turns the power off, etc. the information is loaded again the next time they open it so that they can continue where they left off.

Rather than being hard-coded in the program, the information is user- configurable and typically stored in a plain text format, in this case JSON.

Source: Configuration file[1]

Tip: For the most part, you don't even need to worry about JSON, you just assign your data to the context manager's attributes and JSON Config takes care of the rest.

3.2 Most Interesting Programs Need Some Kind of Configuration

There's no defined standard on how config files should work, *what goes into a configuration file is entirely up to the whim of the developer*, but here are some examples:

Content Management Systems That need store information about where the database server is (the hostname) and how to login (username and password.)

Proprietary Software That need to record whether the software was already registered (the serial key.)

Scientific Software That need to store the path to BLAS libraries.

And the list goes on, and on ....

Source: Configuration files in Python[2]

[1] https://en.wikipedia.org/wiki/Configuration_file
[2] https://martin-thoma.com/configuration-files-in-python

3.2.1 Author's Notes

What is the author storing in the JSON config file? Training set data, information about RESTful API's (authentication information, endpoints, parameters, data extraction & transformation rules, etc.), and much more. It's part of the JSON Transformations project which is the tip of the iceberg for a research project he's working on.

3.2.2 References

CHAPTER 4

Serialization

4.1 Definition

A number of general-purpose serialization formats exist that can represent complex data structures in an easily stored format, and these are often used as a basis for configuration files, *particularly in open-source and platform-neutral software applications and libraries*. The specifications describing these formats are routinely made available to the public, thus increasing the availability of parsers and emitters across programming languages. Examples include: XML, YAML and JSON.

This project is ultra-lite, and is focused solely on JSON; this was a deliberate design decision. For anyone interested in working with the other formats check out the *dynaconf project*.

4.2 Python Data Type Conversions

The *JSON decoder performs the following translations*:

Python	JSON
dict	object
list	array
str	string
int	number (int)
float	number (real)
True	true
False	false
None	null

4.2.1 Want to Learn More about Serialization?

Here are some additional resources:

- **'How to write a JSON configuration file'_**
- *Serialization and deserialization*

4.2.2 References

JSON decoder performs the following translations: https://docs.python.org/3.6/library/json.html

Serialization and deserialization: https://code.tutsplus.com/tutorials/serialization-and-deserialization-of-python-objects-part-1–cms-26183

dynaconf project: https://github.com/rochacbruno/dynaconf

CHAPTER 5

Configuration File Locations

Click[1] *is the package used to determine the default application directory.*

The default behavior is to return whatever is most appropriate for the operating system. To give you an idea, an app called `Foo Bar` would likely return the following:

```
Mac OS X:
~/Library/Application Support/Foo Bar

Mac OS X (POSIX):
~/.foo-bar

Unix:
~/.config/foo-bar

Unix (POSIX):
~/.foo-bar

Win XP (roaming):
C:\Documents and Settings\<user>\Local Settings\Application Data\Foo Bar

Win XP (not roaming):
C:\Documents and Settings\<user>\Application Data\Foo Bar

Win 7 (roaming):
C:\Users\<user>\AppData\Roaming\Foo Bar

Win 7 (not roaming):
C:\Users\<user>\AppData\Local\Foo Bar
```

For additional information visit:

[1] http://click.pocoo.org/5/utils/

5.1 App Dir Options

app_name (required) The `app_name` should be properly capitalized and can contain whitespace. See file location patterns below.

roaming (default: True) Controls if the folder should be roaming or not on Windows; has no effect otherwise.

force_posix (default: False) If this is set to *True* then on any POSIX system the folder will be stored in the home folder with a leading dot instead of the XDG config home or darwin's application support folder.

5.2 The Three (3) File Location Patterns

app_name *app_name is a required argument*

```
with Config('app_name') as data:
    cfg.data = 'Any JSON serializable object ...'
```

The default configuration filename is 'config.json'.

The destination would be: *{click.get_app_dir()}/app_name/config.json*

app_name + cfg_name

```
with Config('app_name', cfg_name='example.json') as data:
    cfg.data = 'Any JSON serializable object ...'
```

The destination would be:

{click.get_app_dir()}/app_name/example.json

app_name w/ path separator (i.e. an explicit filename)

```
with Config('../example.json') as data:
    cfg.data = 'Any JSON serializable object ...'
```

The destination would literally be:

../example.json

5.3 Missing Directories

If any directories in the path are missing JsonConfig will automatically attempt to create them.

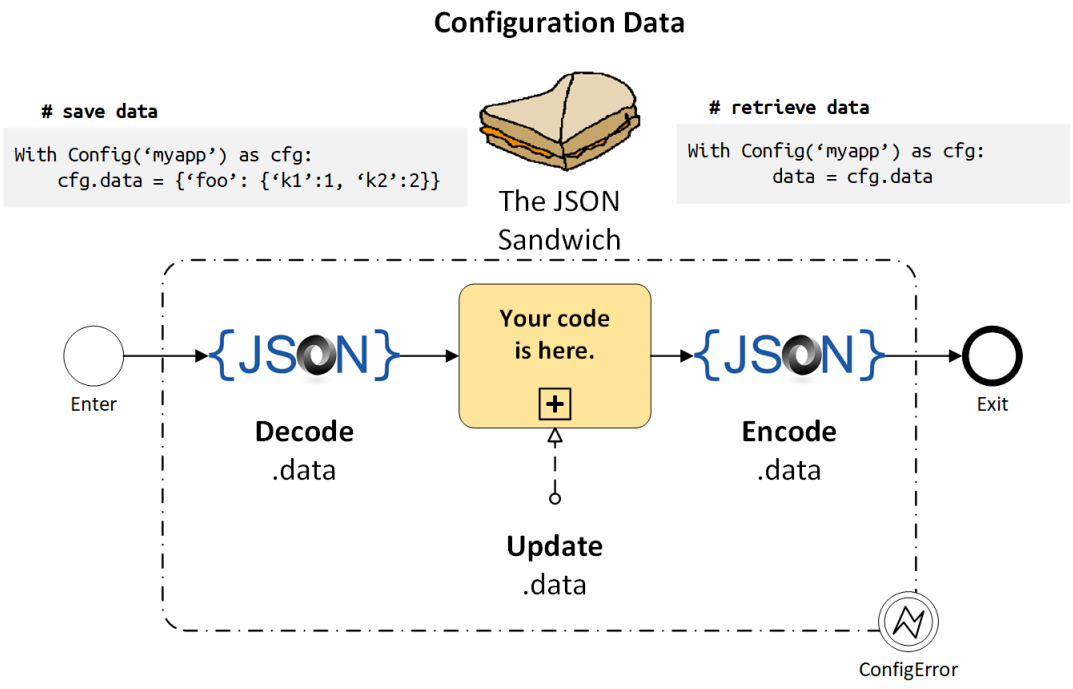
5.4 Missing Files

If a configuration file is missing it will automatically create it.

5.4.1 References

CHAPTER 6

The JSON Sandwich




```
from jsonconfig import Config

with Config('myapp') as cfg:
    cfg.data = {'debug': True}
```

Configuration data is stored in the context manager's 'data' attribute.

cfgfile is the filename; see the Configuration File Locations section to learn how it's constructed:

Your data is in the middle of the sandwich . . .

1. **ENTER**: `.data = json.load(cfgfile)`
2. *your code is here*
3. **EXIT**: `json.dump(cfgfile, .data)`

That's really all there is to it.

The Config context manager's *mode* parameter determines whether the sandwich bread on the top, bottom, neither or both top & bottom.

None No access. Skip step 1 & 3. Do not read or write the data and do not expose the *data* attributed. This is useful is you are only interested in encrypted data and/or environment variables.

```
with Config('myapp', None) as cfg:
    pass
```

r Read-only mode. Skip step 1, set *data = {}*. Of course you can overwrite the data with any JSON serializable object, or apply one of Boxes data access wrappers, but by default it is a standard Python dict.

```
with Config('myapp', 'r') as cfg:
    configuration_data = cfg.data
```

w Write-only mode. Skip step 3. You update the data and it will serialize it as JSON and save it to the config file.

```
with Config('myapp', 'w') as cfg:
    cfg.data = {'debug': True}
```

+ Read & write mode. This is the default. JsonConfig reads in your data and takes a snapshot (deepcopy) of it. You then update the data and then when it exits in step 3 (the context manager's __exit__ method) it will compare the current *data* contents to the snapshot; if the data changed it will serialize the contents of the *data* attribute and save it to the configuration file.

```
with Config('myapp') as cfg:
    cfg.data.update({'width': 80})
```

Note: The above rules and automatic JSON encoding and decoding only apply to the *data* attribute.

The *pwd* (encrypted data) and *env* (environment variables) are updated as soon as you set them and retrieved on demand; they are strings by default. If needed, they can optionally be serialized as JSON using the to_json and from_json helper functions.

All about Python context managers . . .

- https://dbader.org/blog/python-context-managers-and-with-statement
- https://pymotw.com/3/contextlib/
- https://jeffknupp.com/blog/2016/03/07/python-with-context-managers
- http://book.pythontips.com/en/latest/context_managers.html

CHAPTER 7

Data Served in the Wrapper of Your Choice

Box_ *is the package used to handle data access.*

Data wrappers are purely optional, and are designed solely for the purpose of making Mapping/dictionary keys easier to access and improving the readability of your code. If you are not storing Mappings/Dictionaries then you can skip this section.

7.1 Shortcuts

BoxConfig A shortcut for *Config('myapp', box=True)*. It converts the config data attribute into a dictionary that allows both dictionary key and attribute-style access.

```
with BoxConfig('myapp') as cfg:
    cfg.data.widget = {}
    cfg.data.widget.window = {}
    cfg.data.widget.debug = True
    cfg.data.widget.window.title = 'Sample Konfabulator Widget'
    cfg.data.widget.window.width = 500

with BoxConfig('myapp') as cfg:
    assert cfg.data.widget.debug is True
    assert cfg.data.widget.window.title == 'Sample Konfabulator Widget'
    assert cfg.data.widget.window.width == 500
```

FrozenBox A shortcut for *Config('myapp', frozen_box=True)*. Same as *BoxConfig* except that it is read-only. It will not allow updates to the data and will not write back to the configuration file when exitting the context manager.

```
with FrozenBox('myapp') as cfg:
    cfg.debug = True
```

DefaultBox A shortcut for *Config('myapp', default_box=True)*. Acts like a recursive default dict. It automatically creates missing keys.

```
with DefaultBox('myapp') as cfg:
    cfg.data['widget']['debug'] = True
    cfg.data['widget']['window']['title'] = 'Sample Konfabulator Widget'
    cfg.data['widget']['window']['width'] = 800

with DefaultBox('myapp') as cfg:
    assert cfg.data['widget']['debug'] is True
    title = cfg.data['widget']['window']['title']
    assert title == 'Sample Konfabulator Widget'
    assert cfg.data['widget']['window']['width'] == 800
```

7.2 Data Conversion Helpers

The following conversion functions are provided as shortcuts:

BOXED A shorcut for box.Box(self.data). Converts data attribute to a Box object.

FROZEN A shorcut for box.Box(self.data, frozen_box=True). Converts data attribute to a Frozen-Box object.

NESTED A shorcut for box.Box(self.data, default_box=True). Converts data attribute to a Default-Box object.

DATA CONVERSION `BOXED`, `FROZEN` and `NESTED` are all subclasses of dicts or defaultdicts. You can convert back-and-forth between any of them at any time.

OTHER TYPES To return to a standard dict just use *dict(cfg.data)* where *cfg* is your context manager instance. It's just plain Python, just about anything you can do to Python objects you can do to the data attrribute.

7.3 Advanced Usage

JSON Config will pass any valid keyword arguments that box.Box accepts

```
with Config('myapp', camel_killer_box=True) as cfg:
    result = cfg.data
```

See Box's documentation for additional information.

7.3.1 References

All about Python context managers . . .

- https://dbader.org/blog/python-context-managers-and-with-statement
- https://pymotw.com/3/contextlib/
- https://jeffknupp.com/blog/2016/03/07/python-with-context-managers
- http://book.pythontips.com/en/latest/context_managers.html

CHAPTER 8

Encryped Data

Keyring[1] *is the package used to manage encryption.*

To save a secret . . .

```
from jsonconfig import Config

with Config('myapp') as cfg:
    cfg.pwd.some_user = 'some value'
```

To retrieve the secret . . .

```
with Config('myapp') as cfg:
    password = cfg.pwd.some_user
```

8.1 Default Behavior

The default behavior is to select the most secure backend supported by the user's platform. To give you an idea, the following Keyring backends would likely be returned:

Mac OS X: Keychain[2]

Unix (with secretstorage installed): Freedesktop Secret Service[3]

Unix (with dbus installed): kwallet[4]

Windows: Windows Credential Locker[5]

[1] https://github.com/jaraco/keyring
[2] https://en.wikipedia.org/wiki/Keychain_%28software%29
[3] http://standards.freedesktop.org/secret-service
[4] https://en.wikipedia.org/wiki/KWallet
[5] https://technet.microsoft.com/en-us/library/jj554668.aspx

8.2 How-to Set the Service Name

You can think of the service name as the folder where Keyring stores the key/value pair. By default the service name is set the current logged in *username* + '_' + *app_name*. You can override this behavior by explicitly setting the *service_name* in the context manager.

```
with Config('my_app_name', service_name='my_service_name') as cfg:
    cfg.pwd.secret = 'Open Seasame!'
```

8.3 Enabling and Disabling the Keyring

The *keyring* keyword argument controls this.

True This is the default. Enable Keyring and use the default backend.

False Disable the Keyring. The Keyring will not be initalized and the *pwd* attribute will not be available.

```
with Config('myapp', keyring=False) as cfg:
    cfg.data = 'Some value'
```

KeyringConfig This shortcut will enable Keyring and disable data configurations. The *data* attributed will not be available.

from jsonconfig import Keyring

with Keyring('myapp') as vault: vault.pwd.key1 = 'a secret' vault.pwd.key2 = 'another secret'

8.4 Manually Setting the Keyring Backend

Of course, you or the user are free to override the defaults. The user can also change their Keyring backend preferences system-wide from the command-line or via configuration files. JSON Config will then use the user's preferred Keyring backend unless told otherwise.

8.4.1 From the Command Line

```
$ keyring set system username
<enter hidden password for 'username' in 'system'>

$ keyring get system username
password
```

8.4.2 From inside JSON Config

keyring.backends The keyring option accepts a *keyring.backends* class.

```
import keyring.backends

from jsonconfig import Config

backend = keyring.backends.Windows.WinVaultKeyring
```

(continues on next page)

(continued from previous page)

```
with Config('myapp', keyring=backend) as cfg:
    cfg.pwd.some_key = 'a secret'
```

Keyring Backend Name The keyring option accepts a keyring backend name.

```
import keyring.backends

from jsonconfig import Config

with Config('myapp', keyring='WinVaultKeyring') as cfg:
    cfg.pwd.some_key = 'a secret'
```

Valid Keyring names are:

- OS_X
- WIndows
- kwallet
- SecretService

8.5 How it Works

Keyring describes setting a password as follows: *set_password(service, username, password)*. *Username* and *password* do not have to contain user names and password, they are not special; JSON Config treats *username* and *password* as *key* and *value*.

When you set a *pwd* key to a value it calls *set_password(service_name, key, value)*.

When you get a value from a *pwd* key it calls *get_password(service_name, key)*.

8.6 References

CHAPTER 9

Environment Variables

JSON Config provides both dictionary-key and attribute-style access to environment variables via os.environ.

```
from jsonconfig import Environ

with Environ('myapp') as cfg:
    cfg.env.a_variable = 'some value'
```

Environ is a shortcut to Config(mode=None, keyring=False). When mode is None it disables reading & writing to the configuration file and the `data` attribute is not available. When keyring is False it bypasses the Keyring service and the `pwd` attribute is not available.

Note: The app_name for Environ is a required argument, but it doesn't doesn't do anything.

9.1 Attribute & Dictionary Key Style Access

Nested keys are not permitted with environment variables.

Environment variables are accessible through dict style keys and attribute- style notation. When you assign a value to a attribute or a key it will update the environment variable in real-time.

When getting a value if a key is not found it will return `None`.

```
with Environ('myapp') as cfg:
    var = cfg.env.a_variable
```

– or –

```
with Environ('myapp') as cfg:
    var = cfg.env['a_variable']
```

The *env* attribute has most of the attributes and methods associated with dictionaries. For example:

```
with Environ('myapp') as cfg:
    print(cfg.env.keys())
```

9.2 Getpass Helper Function

There is a helper function called `getpass()` that will allow you to prompt the user for a passowrd.

For example to prompt for a password in an environment variable is not set ...

```
with Environ('myapp') as cfg:
    password = cfg.env.mypassword or getpass
```

Or to check the keyring vault first, if it's not set there then check the environment variables, if it is not set there then prompt for a password.

```
with Config('myapp', mode=None) as cfg:
    password = cfg.pwd.mypassword or cfg.env.mypassword or getpass
```

You can also set it if it's not set:

```
with Config('myapp', mode=None) as cfg:
    cfg.env.mypassword = cfg.pwd.mypassword or cfg.env.mypassword or getpass
```

9.3 JSON Helper Functions

It's not usually used with environment variables, but it is possible to store JSON serialized objects in environment variables.

```
with Environ('myapp', mode=None) as cfg:
    cfg.env.settings = to_json({'debug': True, 'width': 80})
```

To retrieve it ...

```
with Environ('myapp', mode=None) as cfg:
    settings = from_json(cfg.env.settings)
```

CHAPTER 10

Exception Handling

The exceptions are designed so that you can wrap the whole Config with a single try, or choose your level or granularity.

```
try:
    with Config('myapp') as cfg:
        cfg.data = {"mongodb": {"host": "localhost", "port": 27017}}
except JsonConfigError as e:
    e.show()
```

JsonConfigError(Exception) Base Exception. Use the function show(exitstatus=1) to display the error. If exitstatus is not equal to zero then exit the program after displaying the error.

FileError(JsonConfigError, EnvironmentError) File I/O error or O.S. related issue.

FileEncodeError(JsonConfigError, ValueError) When reading/writing to config file with errors='strict'.

JsonEncodeError(JsonConfigError, TypeError) Not JSON seriable.

JsonDecodeError(JsonConfigError, ValueError) Not valid JSON.

SetEnvironVarError(JsonConfigError, TypeError) Unable to set environment variable.

DeleteEnvironVarError(JsonConfigError, KeyError) Unable to delete environment variable.

SetPasswordError(JsonConfigError, keyring.errors.PasswordSetError) Unable to set password.

DeletePasswordError(JsonConfigError, keyring.errors.PasswordDeleteError) Unable to delete password.

KeyringNameError(JsonConfigError) Invalid Keyring Backend Name.

CHAPTER 11

Pass-Through Keyword Arguments

JSON Config stays out of your way as a Python developer. There's no magic, you can use Config context manager to pass any valid keyword arguments to the functions below.

if `readable` or `writable`:

1. *click.get_app_dir*

if `readable`:

2. *io.open if PY2 else open*
3. *json.load*

if `box`:

4. *box.Box*

if `writable`:

5. *io.open if PY2 else open*
6. *json.dump*

The only limitation is when there is when there is a name collision, currently the only known collision is the `cls` argument, which is used in both json.load and json.dump.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search