cincoconfig Documentation

Release 0.9.0

Adam Meily

Aug 06, 2023

CONTENTS:

CHAPTER

ONE

RECIPES

1.1 Validate Configuration On Load

The *Schema* object has a *validator()* decorator that registers a method as a validator. Whenever the configuration is loaded, the schema's validators are run against the configuration. Custom config validators can perform advanced validations, like the following:

```
schema = Schema()
schema.x = IntField()
schema.y = IntField()
schema.db.username = StringField()
schema.db.password = StringField()

@validator(schema)
def validate_x_lt_y(cfg):
    # validates that x < y
    if cfg.x and cfg.y and cfg.x >= cfg.y:
        raise ValueError('x must be less-than y')

@validator(schema.db)
def validate_db_credentials(cfg):
    # validates that if the db username is specified then the password must
    # also be specified.
    if cfg.username and not db.password:
        raise ValueError('db.password is required when username is specified')

config = schema()
config.load('config.json', format='json')  # will call the above validators
```

1.2 Allow Multiple Configuration Files

The *IncludeField* allows users to specify an additional file to parse when loading the config from disk. The `IncludeField` is processes prior to setting parsing any of the configuration values. So, the entire config file and any included config files are combined in memory prior to parsing.

```
schema = Schema()
schema.include = IncludeField()
# other fields
config = schema()
```

(continues on next page)

(continued from previous page)

```
config.load('config.json', format='json')
```

config.json

```
{
    "include": "other_file.json"
}
```

IncludeFields can occur anywhere in a configuration, however, when the included file is processed, it is handled in the same scope as where the IncludeField was defined.

1.3 Dynamically Get and Set Configuration Values

The *Config* class supports getting and setting config values via both direct attribute access and the `__getitem__` and `__setitem__` protocols. The `__getitem__` and `__setitem__` methods have the added benefit of supporting getting and setting nested config values.

```
schema = Schema()
schema.x = IntField()
schema.db.port = IntField(default=27017)
schema.db.host = HostnameField(default='127.0.0.1', allow_ipv4=True)

config = schema()
config.load('config.json', format='json')

#
# get the set port
# equivalent to:
#
#     print(config.db.port)
#
print(config['db.port'])

#
# set the hostname
# equivalent to:
#
#    config.db.host = 'db.example.com'
#
config['db.host'] = 'db.example.com'
```

Using `__getitem__` and `__setitem__` is useful in situations where you need dynamic programmatic access to the configuration values, such as supporting a generic REST API to interact with the configuration.

1.4 List Field of Complex Types

The *ListField* performs value validation for each item by accepting either a *Field* or *Schema*. Consider the example of a list of webhooks.

```
webhook_schema = Schema()
webhook_schema.url = UrlField(required=True)
webhook_schema.verify_ssl = BoolField(default=True)

schema = Schema()
schema.issue_webhooks = ListField(webhook_schema)
schema.merge_request_webhooks = ListField(webhook_schema)

config = schema()

wh = webhook_schema()
wh.url = 'https://google.com'
config.issue_webhooks.append(wh)
```

Here, the webhook schema is usable across multiple configurations. As seen here, it is not very intuitive to create reusable configuration items. The *make_type()* method is designed to make working with these reusable configurations easier. make_type creates a new type, inheriting from *Config* that is more Pythonic.

```
webhook_schema = Schema()
webhook_schema.url = UrlField(required=True)
webhook_schema.verify_ssl = BoolField(default=True)
WebHook = make_type(webhook_schema, 'WebHook')  # WebHook is now a new type

schema = Schema()
schema.issue_webhooks = ListField(webhook_schema)
schema.merge_request_webhooks = ListField(webhook_schema)

config = schema()

wh = WebHook(url='https://google.com')
config.issue_webhooks.append(wh)
```

CHAPTER

TWO

CONFIGS

class cincoconfig.**Schema**(*key=None*, *name=None*, *dynamic=False*, *env=None*, *schema=None*)

A config schema containing all available configuration options. A schema's fields and hierarchy are built dynamically.

```
schema = Schema()
schema.mode = ApplicationModeField(default='production', required=True)
schema.http.port = PortField(default=8080)
# the previous line implicitly performs "schema.http = Schema(key='http')"
schema.http.host = IPv4Address(default='127.0.0.1')
```

Accessing a field that does not exist, such as `schema.http` in the above code, dynamically creates and adds a new `Schema`.

Once a schema is completely defined, a *Config* is created by calling the schema. The config is populate with the default values specified for each field and can then load the configuration from a file.

Parameters

- **key** (`Optional`[`str`]) – schema field key
- **dynamic** (`bool`) – configurations created from this schema are dynamic and can add fields not originally in the schema *_key*
- **env** (`Union`[`bool`, `str`, `None`]) – the environment variable prefix for this schema and all children schemas, for information, see *Field Environment Variables*

_add_field(*name*, *field*)

Add a field to the schema. This method will call `field.__setkey__(self, key)`.

Return type
`BaseField`

Returns
the added field (`field`)

_get_field(*key*)

Return type
`Optional`[`BaseField`]

Returns
a field from the schema

__setattr__(*name*, *value*)

Parameters

- **name** (str) – attribute name
- **value** (Any) – field or schema to add to the schema

Return type
Any

__getattr__(*name*)

Retrieve a field by key or create a new Schema if the field doesn't exist.

Parameters
name (str) – field or schema key

Return type
BaseField

__setitem__(*name*, *value*)

Return type
BaseField

Returns

field, equivalent to `setattr(schema, key)`, however his method handles setting nested values. For example:

```
>>> schema = Schema()
>>> schema.x = Schema()
>>> schema['x.y'] = IntField(default=10)
>>> print(schema.x.y)
IntField(key='y', ...)
```

__getitem__(*key*)

Return type
BaseField

Returns

field, equivalent to `getattr(schema, key)`, however his method handles retrieving nested values. For example:

```
>>> schema = Schema()
>>> schema.x.y = IntField(default=10)
>>> print(schema['x.y'])
IntField(key='y', ...)
```

__iter__()

Iterate over schema fields, produces as a list of tuples `(key, field)`.

Return type
Iterator[Tuple[str, BaseField]]

__call__(*parent=None*, ***data*)

Compile the schema into an initial config with default values set.

_ref_path

Get the full reference path to the schema field. For example:

```
>>> schema = Schema()
>>> schema.x.y.z = Field()
>>> schema.x.y.z._ref_path
'x.y.z'
```

generate_argparse_parser(***parser_kwargs*)

(Deprecated, will be removed in v1.0.0) generate an ArgumentParser for the schema. Use *generate_argparse_parser()*.

Return type
ArgumentParser

Returns
an ArgumentParser containing arguments that match the schema's fields

get_all_fields()

(Deprecated, will be removed in v1.0.0) get all the fields in the configuration. Use *get_all_fields()*.

Return type
List[Tuple[str, *Schema*, BaseField]]

Returns
a list of tuples with (key, schema, field)

instance_method(*key*)

(Deprecated, will be removed in v1.0.0) decorator to register an instance method with the schema. Use *instance_method()*.

Return type
Callable[[*Config*], None]

make_type(*name*, *module=None*, *key_filename=None*)

(Deprecated, will be removed in v1.0.0) create a new type from the schema. Use *make_type()*.

Return type
Type[*ConfigType*]

validator(*func*)

(Deprecated, will be removed in v1.0.0) decorator to register a validator method with the schema. Use *validator()*.

Return type
Callable[[*Config*], None]

class cincoconfig.**Config**(*schema*, *parent=None*, *key_filename=None*, ***data*)

A configuration.

Parsing and serializing the configuration is done via an intermediary object, a tree (dict) containing only basic (serializable) values (see Field to_basic()).

When saving, the config will convert the current config values to a tree and then pass the tree to the specified format. When loading, the file content's will be passed to the formatter, which will return a basic tree that the config will validate and convert to actual config values.

The initial config will be populated with default values, specified in each Field's *default* value. If the configuration needs to be initialized programmatically, prior to loading from a file, the *load_tree()* method can be used to load a basic tree.

```
schema = Schema()
schema.port = PortField()
schema.host = HostnameField()

config = schema()
# We didn't specify any default values, load from a dict

config.load_tree({
    'port': 8080,
    'host': '127.0.0.1'
})

# Loading an initial tree above is essentially equivalent to:
#
# schema = Schema()
# schema.port = PortField(default=8080)
# schema.host = HostnameField(default='127.0.0.1')
# config = schema()
```

Each config object can have an associated *cincoconfig.KeyFile*, passed in the constructor as `key_filename`. If the configuration file doesn't have a key file path set, the config object will use the parent config's key file. Requesting a key file will bubble up to the first config object that has the key filename set and, if no config has a keyfile, the default path will be used, `DEFAULT_CINCOKEY_FILEPATH`.

Parameters

- **schema** (*Schema*) – backing schema, stored as *_schema*
- **parent** (Optional[*Config*]) – parent config instance, only set when this config is a field of another config, stored as *_parent*
- **key_filename** (Optional[str]) – path to key file
- **data** – configuration values

_get_field(*key*)

Return type
Optional[BaseField]

Returns
a field from the schema or the dynamically added field if the schema is dynamic.

_set_value(*key*, *value*)

Set a configuration value. This method passes the value through the field validation chain and then calls the target field's `__setval__()` to actually set the value.

Any exception that is raised by the field validation will be wrapped in an `ValidationError` and raised again.

Parameters

- **name** – field key
- **value** (Any) – value to validate and set

Raises
ValidationError – setting the value failed

Return type
Any

_set_default_value(*key*, *value*)

Set a default value without performing any validation. The value is set and the field is marked as having the initial default value.

Parameters

- **key** (str) – field key
- **value** (Any) – field default value

Return type
None

_get_value(*key*)

Return type
Any

__setattr__(*name*, *value*)

Validate a configuration value and set it.

Parameters

- **name** (str) – field key
- **value** (Any) – value

Return type
Any

__getattr__(*name*)

Retrieve a config value.

Parameters
name (str) – field key

Return type
Any

__setitem__(*key*, *value*)

Set a field value, equivalent to `setattr(config, key)`, however this method handles setting nested values. For example:

```
>>> schema = Schema()
>>> schema.x.y = IntField(default=10)
>>> config = schema()
>>> config['x.y'] = 20
>>> print(config.x.y)
20
```

Return type
Any

__getitem__(*key*)

Return type
Any

Returns

field value, equivalent to `getattr(config, key)`, however his method handles retrieving nested values. For example:

```
>>> schema = Schema()
>>> schema.x.y = IntField(default=10)
>>> config = schema()
>>> print(config['x.y'])
10
```

__iter__()

Return type
Iterator[Tuple[str, Any]]

__contains__(*key*)

Check if key is in the configuration. This method handles checking nested values. For example:

```
>>> schema = Schema()
>>> schema.x.y = IntField(default=10)
>>> config = schema()
>>> 'x.y' in config
True
```

Return type
bool

_ref_path

Returns
the full reference path to the configuration

_keyfile

Returns
the config's encryption key file (if not set, get the parent config's key file)

_key_filename

Returns
the path to the cinco encryption key file (if not set, get the parent config's key filename)

cmdline_args_override(*args*, *ignore=None*)

(Deprecated, will be removed in v1.0.0) override configuration values from the command line arguments. Use *cmdline_args_override()*.

Parameters

- **args** (Namespace) – parsed arguments
- **ignore** (Union[str, List[str], None]) – list of field keys to ignore

Return type
None

dumps(*format*, *virtual=False*, *sensitive_mask=None*, ***kwargs*)

Serialize the configuration to a string with the specified format.

Parameters

- **format** (str) – output format
- **virtual** (bool) – include virtual fields in the output
- **sensitive_mask** (Optional[str]) – replace secure values, see *to_tree()*
- **kwargs** – additional keyword arguments to pass to the formatter's __init__()

Return type
bytes

Returns
serialized configuration file content

property full_path: str

(Deprecated, will be removed in v1.0.0) get the full path to the configuration :returns: the full path to this configuration

load(*filename*, *format*)

Load the configuration from a file.

Parameters

- **filename** (str) – source filename
- **format** (str) – source format

load_tree(*tree*, *validate=True*)

Load a tree and then validate the values.

Parameters
tree (dict) – a basic value tree

Return type
None

loads(*content*, *format*, ***kwargs*)

Load a configuration from a str or bytes and process any *IncludeField*.

Parameters

- **content** (Union[str, bytes]) – configuration content
- **format** (str) – content format
- **kwargs** – additional keyword arguments to pass to the formatter's __init__()

save(*filename*, *format*, ***kwargs*)

Save the configuration to a file. Additional keyword arguments are passed to *dumps()*.

Parameters

- **filename** (str) – destination file path
- **format** (str) – output format

Param
additional keyword arguments for *dumps()*

to_tree(*virtual=False*, *sensitive_mask=None*)

Convert the configuration values to a tree.

The *sensitive_mask* parameter is an optional string that will replace sensitive values in the tree.

- None (default) - include the value as-is in the tree

- `len(sensitive_mask) == 1` (single character) - replace every character with the `sensitive_mask` character. `value = sensitive_mask * len(value)`
- `len(sensitive_mask) != 1` (empty or multi character string) - replace the entire value with the `sensitive_mask`.

Parameters

- **virtual** (bool) – include virtual field values in the tree
- **sensitive_mask** (Optional[str]) – mask secure values with a string

Return type
dict

Returns
the basic tree containing all set values

validate(*collect_errors=False*)

Perform validation on the entire config.

Return type
List[ValidationError]

class cincoconfig.**ConfigType**(*parent=None*, ***kwargs*)

A base class for configuration types. A subclass of `ConfigType` is returned by the *make_type()* function.

Parameters
parent (Optional[*Config*]) – parent configuration

CHAPTER

THREE

FIELDS

class cincoconfig.**Field**(*, *key=None*, *schema=None*, *name=None*, *required=False*, *default=None*, *validator=None*, *sensitive=False*, *description=None*, *help=None*, *env=None*)

The base configuration field. Fields provide validation and the mechanisms to retrieve and set values from a *Config*. Field's are composable and reusable so they should not store state or store the field value.

Validation errors should raise a `ValueError` exception with a brief message.

There are three steps to validating a value:

1. *validate()* - checks the value against the *required* parameter and then calls:
2. *_validate()* - validate function that is implemented in subclasses of `Field`
3. `Field.validator` - custom validator method specified when the field is created

The pseudo code for the *validate()* function:

```
def validate(self, cfg, value):
    if value is None and self.required:
        raise ValueError

    value = self._validate(cfg, value)
    if self.validator:
        value = self.validate(cfg, value)
    return value
```

Since each function in the validation chain returns the value, each validator can transform the value. For example, the *BoolField* `_validate` method converts the string value to a `bool`.

Each Field has the following lifecycle:

1. `__init__` - created by the application
2. *__setkey__()* - the field is added to a *Schema*
3. ***__setdefault__()* - the field is added to a config and the config is populated with the** default value

Whenever a config value is set, the following methods are called in this order:

1. *validate()* / *_validate()* / `validator` - validation chain
2. *__setval__()* - set a validated value to the config

Finally, whenever code retrieves a config value, the *__getval__()* is called.

Most field subclasses only need to implement the `_validate` method and most do not need to implement the `__setkey__`, `__setdefault__`, `__getval__` and `__setval__` methods, unless the field needs to modify the default behavior of these methods.

The Field `_key` is used to set and reference the value in the config.

Each Field subclass can define a class or instance level `storage_type` which holds the annotation of the value being stored in memory.

Environment Variables

Fields can load their default value from an environment variable. The Schema and Field accept an `env` argument in the constructor that controls whether and how environment variables are loaded. The default behavior is to not load any environment variables and to honor the *`Field.default`* value.

There are two ways to load a field's default value from an environment variable.

- `Schema.env`: Provide `True` or a string.
- `Field.env`: Provide `True` or a string.

When `Schema.env` or `Field.env` is `None` (the default), the environment variable configuration is inherited from the parent schema. A value of `True` will load the the field's default value from an autogenerated environment variable name, based on the field's full path. For example:

```
schema = Schema(env=True)
schema.mode = ApplicationModeField(env="APP_MODE")
schema.port = PortField(env=False)

schema.db.host = HostnameField()

schema.auth = Schema(env="SECRET")
schema.auth.username = StringField()
```

- The top-level schema is configured to autogenerate and load environment variables for all fields.
- `mode` is loaded from the `APP_MODE` environment variable.
- `port` is not loaded from any the environment variable.
- `db.host` is loaded from the `DB_HOST` environment variable.
- The `auth` schema has a environment variable prefix of `SECRET`. All children and nested fields/schemas will start with `SECRET_`.
- The `auth.username` field is loaded from the `SECRET_USERNAME` environment variable.

All builtin Fields accept the following keyword parameters.

Parameters

- **name** (`Optional[str]`) – field friendly name, used for error messages and documentation
- **key** (`Optional[str]`) – the key of the field in the config, this is typically not specified and, instead the *`__setkey__()`* will be called by the config
- **required** (`bool`) – the field is required and a `ValueError` will be raised if the value is `None`
- **default** (`Union[Callable, Any, None]`) – the default value, which can be a called that is invoke with no arguments and should return the default value
- **validator** (`Optional[Callable[[Config, Any], Any]]`) – an additional validator function that is invoked during validation

- **sensitive** (bool) – the field stores a sensitive value
- **help** (Optional[str]) – the field documentation

_validate(*cfg*, *value*)

Subclass validation hook. The default implementation just returns `value` unchanged.

Return type
Any

__setkey__(*schema*, *key*)

Set the field's *_key*, which is called when the field is added to a schema. The default implementation just sets `self._key = key`

Parameters

- **schema** (*Schema*) – the schema the field belongs to
- **key** (str) – the field's unique key

__setdefault__(*cfg*)

Set the default value of the field in the config. This is called when the config is first created.

Parameters
cfg (*Config*) – current config

Return type
None

__getval__(*cfg*)

Retrieve the value from the config. The default implementation retrieves the value from the config by the field *key*.

Parameters
cfg (*Config*) – current config

Return type
Any

Returns
the value stored in the config

__setval__(*cfg*, *value*)

Set the validated value in the config. The default implementation passes the value through the validation chain and then set's the validated value int the config.

Parameters

- **cfg** (*Config*) – current config
- **value** (Any) – value to validated

property default: Any

Returns
the field's default value

property name

Returns
the field's friendly name: `name or key`

property short_help: Optional[str]

A short help description of the field. This is derived from the `help` attribute and is the first paragraph of text in `help`. The intention is that `short_help` can be used for the field description and `help` will have the full documentation. For example:

```
>>> field = Field(help="""
... This is a short description
... that can span multiple lines.
...
... This is more information.
... """)
>>> print(field.short_help)
this is a short description
that can span multiple lines.
```

Returns
the first paragraph of `help`

`to_basic`(*cfg*, *value*)

Convert the Python value to the basic value.

The default implementation just returns `value`. This method is called when the config is saved to a file and will only be called with the value associated with this field.

Parameters

- **cfg** (*Config*) – current config
- **value** (Any) – value to convert to a basic type

Return type
Any

Returns
the converted basic type

`to_python`(*cfg*, *value*)

Convert the basic value to a Python value. Basic values are serializable (ie. not complex types). The following must hold true for config file saving and loading to work:

```
assert field.to_python(field.to_basic(value)) == value
```

The default implementation just returns `value`. This method is called when the config is loaded from a file and will only be called with the value associated with this field.

In general, basic types are any types that can be represented in JSON: string, number, list, dict, boolean.

Parameters

- **cfg** (*Config*) – current config
- **value** (Any) – value to convert to a Python type

Return type
Any

Returns
the converted Python type

validate(*cfg*, *value*)

Start the validation chain and verify that the value is specified if *required=True*.

Parameters

- **cfg** (*Config*) – current config
- **value** (Any) – value to validate

Return type
Any

Returns
the validated value

class cincoconfig.**StringField**(*, *min_len=None*, *max_len=None*, *regex=None*, *choices=None*, *transform_case=None*, *transform_strip=None*, ***kwargs*)

A string field.

The string field can perform transformations on the value prior to validating it if either *transform_case* or *transform_strip* are specified.

Parameters

- **min_len** (Optional[int]) – minimum allowed length
- **max_len** (Optional[int]) – maximum allowed length
- **regex** (Optional[str]) – regex pattern that the value must match
- **choices** (Optional[List[str]]) – list of valid choices
- **transform_case** (Optional[str]) – transform the value's case to either `upper` or `lower` case
- **transform_strip** (Union[bool, str, None]) – strip the value by calling `str.strip()`. Setting this to `True` will call `str.strip()` without any arguments (ie. striping all whitespace characters) and if this is a `str`, then `str.strip()` will be called with `transform_strip`.

storage_type

alias of str

class cincoconfig.**LogLevelField**(*levels=None*, ***kwargs*)

A field representing the Python log level.

Parameters
levels (Optional[List[str]]) – list of log levels. If not specified, the default Python log levels will be used: `debug`, `info`, `warning`, `error`, and `critical`.

storage_type

alias of str

class cincoconfig.**ApplicationModeField**(*modes=None*, *create_helpers=True*, ***kwargs*)

A field representing the application operating mode.

The *create_helpers* parameter will create a boolean *VirtualField* for each `mode` named `is_<mode>_mode`, that returns `True` when the mode is active. When *create_helpers=True* then each mode name must be a valid Python variable name.

Parameters

- **modes** (Optional[List[str]]) – application modes, if not specified the default modes will be used: `production` and `development`

- **create_helpers** (bool) – create helper a bool VirtualField for each mode

storage_type

alias of str

class cincoconfig.**SecureField**(*method='best'*, *sensitive=True*, ***kwargs*)

A secure storage field where the plaintext configuration value is encrypted on disk and decrypted in memory when the configuration file is loaded.

Parameters

method (str) – encryption method, see *_get_provider()*

storage_type

alias of str

class cincoconfig.**IntField**(***kwargs*)

Integer field.

storage_type

alias of int

class cincoconfig.**FloatField**(***kwargs*)

Float field.

storage_type

alias of float

class cincoconfig.**PortField**(***kwargs*)

Network port field.

storage_type

alias of int

class cincoconfig.**IPv4AddressField**(***, *min_len=None*, *max_len=None*, *regex=None*, *choices=None*, *transform_case=None*, *transform_strip=None*, ***kwargs*)

IPv4 address field.

The string field can perform transformations on the value prior to validating it if either *transform_case* or *transform_strip* are specified.

Parameters

- **min_len** (Optional[int]) – minimum allowed length
- **max_len** (Optional[int]) – maximum allowed length
- **regex** (Optional[str]) – regex pattern that the value must match
- **choices** (Optional[List[str]]) – list of valid choices
- **transform_case** (Optional[str]) – transform the value's case to either upper or lower case
- **transform_strip** (Union[bool, str, None]) – strip the value by calling str.strip(). Setting this to True will call str.strip() without any arguments (ie. striping all whitespace characters) and if this is a str, then str.strip() will be called with transform_strip.

storage_type

alias of str

class cincoconfig.**IPv4NetworkField**(*min_prefix_len=None*, *max_prefix_len=None*, ***kwargs*)

IPv4 network field. This field accepts CIDR notation networks in the form of `A.B.C.D/Z`.

Parameters

- **min_prefix_len** (Optional[int]) – minimum subnet prefix length (/X), in bits
- **max_prefix_len** (Optional[int]) – maximum subnet prefix length (/X), in bits

storage_type

alias of str

class cincoconfig.**HostnameField**(***, *allow_ipv4=True*, *resolve=False*, ***kwargs*)

A field representing a network hostname or, optionally, a network address.

Parameters

- **allow_ipv4** (bool) – allow both a hostname and an IPv4 address
- **resolve** (bool) – resolve hostnames to their IPv4 address and raise a ValueError if the resolution fails

storage_type

alias of str

class cincoconfig.**FilenameField**(***, *exists=None*, *startdir=None*, ***kwargs*)

A field for representing a filename on disk.

The *exists* parameter can be set to one of the following values:

- `None` - don't check file's existence
- `False` - validate that the filename does not exist
- `True` - validate that the filename does exist
- `"dir"` - validate that the filename is a directory that exists
- `"file"` - validate that the filename is a file that exists

The *startdir* parameter, if specified, will resolve filenames starting from a directory and will cause all filenames to be validate to their absolute file path. If not specified, filename's will be resolve relative to `os.getcwd()` and the relative file path will be validated.

Parameters

- **exists** (Union[bool, str, None]) – validate the filename's existence on disk
- **startdir** (Optional[str]) – resolve relative paths to a start directory

storage_type

alias of str

class cincoconfig.**BoolField**(***, *key=None*, *schema=None*, *name=None*, *required=False*, *default=None*, *validator=None*, *sensitive=False*, *description=None*, *help=None*, *env=None*)

A boolean field.

All builtin Fields accept the following keyword parameters.

Parameters

- **name** (Optional[str]) – field friendly name, used for error messages and documentation
- **key** (Optional[str]) – the key of the field in the config, this is typically not specified and, instead the `__setkey__()` will be called by the config

- **required** (bool) – the field is required and a ValueError will be raised if the value is None
- **default** (Union[Callable, Any, None]) – the default value, which can be a called that is invoke with no arguments and should return the default value
- **validator** (Optional[Callable[[*Config*, Any], Any]]) – an additional validator function that is invoked during validation
- **sensitive** (bool) – the field stores a sensitive value
- **help** (Optional[str]) – the field documentation

FALSE_VALUES = ('f', 'false', '0', 'off', 'no', 'n')

Accepted values that evaluate to False

TRUE_VALUES = ('t', 'true', '1', 'on', 'yes', 'y')

Accepted values that evaluate to True

storage_type

alias of bool

class cincoconfig.**FeatureFlagField**(***, *key=None*, *schema=None*, *name=None*, *required=False*, *default=None*, *validator=None*, *sensitive=False*, *description=None*, *help=None*, *env=None*)

Concrete implementation of the feature flag field. When this field's value is set to False, the bound configurations will not perform validation.

All builtin Fields accept the following keyword parameters.

Parameters

- **name** (Optional[str]) – field friendly name, used for error messages and documentation
- **key** (Optional[str]) – the key of the field in the config, this is typically not specified and, instead the `__setkey__()` will be called by the config
- **required** (bool) – the field is required and a ValueError will be raised if the value is None
- **default** (Union[Callable, Any, None]) – the default value, which can be a called that is invoke with no arguments and should return the default value
- **validator** (Optional[Callable[[*Config*, Any], Any]]) – an additional validator function that is invoked during validation
- **sensitive** (bool) – the field stores a sensitive value
- **help** (Optional[str]) – the field documentation

class cincoconfig.**UrlField**(***, *min_len=None*, *max_len=None*, *regex=None*, *choices=None*, *transform_case=None*, *transform_strip=None*, ***kwargs*)

A URL field. Values are validated that they are both a valid URL and contain a valid scheme.

The string field can perform transformations on the value prior to validating it if either *transform_case* or *transform_strip* are specified.

Parameters

- **min_len** (Optional[int]) – minimum allowed length
- **max_len** (Optional[int]) – maximum allowed length
- **regex** (Optional[str]) – regex pattern that the value must match
- **choices** (Optional[List[str]]) – list of valid choices

- **transform_case** (Optional[str]) – transform the value's case to either `upper` or `lower` case
- **transform_strip** (Union[bool, str, None]) – strip the value by calling `str.strip()`. Setting this to `True` will call `str.strip()` without any arguments (ie. striping all whitespace characters) and if this is a `str`, then `str.strip()` will be called with `transform_strip`.

storage_type

alias of `str`

class cincoconfig.**ListField**(*field=None*, ***kwargs*)

A list field that can optionally validate items against a `Field`. If a field is specified, a *ListProxy* will be returned by the `_validate` method, which handles individual item validation.

Specifying *required=True* will cause the field validation to validate that the list is not `None` and is not empty.

Parameters
field (Union[BaseField, Type[*ConfigType*], None]) – Field to validate values against

to_basic(*cfg*, *value*)

Convert to basic type.

Parameters

- **cfg** (*Config*) – current config
- **value** (Union[list, *ListProxy*]) – value to convert

Return type
list

to_python(*cfg*, *value*)

Convert to Pythonic type.

Parameters

- **cfg** (*Config*) – current config
- **value** (list) – basic type value

Return type
Union[list, *ListProxy*]

class cincoconfig.**VirtualField**(*getter*, *setter=None*, ***kwargs*)

A calculated, readonly field that is not read from or written to a configuration file.

Parameters

- **getter** (Callable[[*Config*], Any]) – a callable that is called whenever the value is retrieved, the callable will receive a single argument: the current *Config*.
- **setter** (Optional[Callable[[*Config*, Any], Any]]) – a callable that is called whenever the value is set, the callable will receive two arguments: `config, value`, the current *Config* and the value being set

class cincoconfig.**DictField**(*key_field=None*, *value_field=None*, ***kwargs*)

A generic `dict` field that optionally validates keys and values. The `key_field` and `value_field` parameters control whether and how the dictionary keys and values are validated, respectively. Setting these will cause the internal representation to be stored in a *DictProxy* which handles the validation operations.

Specifying *required=True* will cause the field validation to validate that the `dict` is not `None` and is not empty.

storage_type

alias of dict

to_basic(*cfg*, *value*)

Convert to basic type.

Parameters

- **cfg** (*Config*) – current config
- **value** (Union[dict, *DictProxy*]) – value to convert

Return type
dict

to_python(*cfg*, *value*)

Convert to Pythonic type.

Parameters

- **cfg** (*Config*) – current config
- **value** (dict) – basic type value

Return type
Union[dict, *DictProxy*]

class cincoconfig.**BytesField**(*encoding='base64'*, ***kwargs*)

Store binary data in an encoded string.

Parameters
encoding (str) – binary data encoding, must be one of *ENCODINGS*

ENCODINGS = ('base64', 'hex')

Available encodings: base64 and hex

storage_type

alias of bytes

to_basic(*cfg*, *value*)

Return type
str

Returns
the encoded binary data

to_python(*cfg*, *value*)

Return type
Optional[bytes]

Returns
the decoded binary data

class cincoconfig.**IncludeField**(*startdir=None*, ***kwargs*)

A special field that can include another configuration file when loading from disk. Included files are in the same scope as where the include field is defined for example:

```
# file1.yaml
db:
  include: "db.yaml"
include: "core.yaml"

# db.yaml
host: "0.0.0.0"
port: 27017

# core.yaml
mode: "production"
ssl: true
```

The final parsed configuration would be equivalent to:

```
db:
  host: "0.0.0.0"
  port: 27017

mode: "production"
ssl: true
```

Included files must be in the same configuration file format as their parent file. So, if the base configuration file is stored in JSON then every included file must also be in JSON.

Cincoconfig does not track which configuration file set which field(s). When a config file is saved back to disk, it will be the entire configuration, even if it was originally defined across multiple included files.

Parameters
startdir (Optional[str]) – resolve relative include paths to a start directory

combine_trees(*base*, *child*)

An extension to dict.update() but properly handles nested *dict* objects.

Parameters

- **base** (dict) – base tree to extend
- **child** (dict) – child tree to apply onto base

Return type
dict

Returns
the new combined dict

include(*config*, *fmt*, *filename*, *base*)

Include a configuration file and combine it with an already parsed basic value tree. Values defined in the included file will overwrite values in the base tree. Nested trees (dict objects) will be combined using a dict.update() like method, *combine_trees()*.

Parameters

- **config** (*Config*) – configuration object
- **fmt** (*ConfigFormat*) – configuration file format that will parse the included file
- **filename** (str) – included file path
- **base** (dict) – base config value tree

Return type
dict

Returns
the new basic value tree containing the base tree and the included tree

3.1 Secure Fields

The following fields provide secure configuration option storage and challenges.

class cincoconfig.**ChallengeField**(*hash_algorithm='sha256'*, ***kwargs*)

A field whose value is securely stored as a hash (*DigestValue*). This field can be used as a secure method of password storage and comparison, since the password is only stored in hashed form and not in plaintext. A digest value is pair of salt and `hash(salt + plaintext)` values.

Values are stored in memory as *DigestValue* instances. For example:

```
>>> schema = Schema()
>>> schema.password = ChallengeField('md5')
>>> cfg = schema()
>>> cfg.password = "Hello"
>>> print(type(cfg.password))
<class 'cincoconfig.fields.DigestValue'>

>>> print(cfg.password)
Yt4Qm5cC9FoRSdU3Ly7B7A==:+GXXhO36XvJ446fqXYJ+1w==

>>> cfg.password.digest
b'øe×íú^òxã§ê]~×'
```

The `default` value of a challenge field can be either:

- A plaintext string. In this case, the salt will be randomly generated.
- A *DigestValue* instance.

When the default value is a string, the salt will change between application executions. For example:

```
>>> schema = Schema()
>>> schema.password = ChallengeField('md5', default='hello')
>>> cfg = schema()
# First time application executes
>>> print(cfg.password)
Yt4Qm5cC9FoRSdU3Ly7B7A==:+GXXhO36XvJ446fqXYJ+1w==

# Second time application executes
>>> print(cfg.password)
c2MPwSJw1QYMOcE2O+cVFA==:JSNbBj3wCgh7alFM7l0geg==
```

Digest values are saved to disk as a `dict` containing two keys:

- `salt` - base64 encoded salt
- `digest` - base64 encoded digest

The challenge field supports loading plaintext string values from the configuration file. So, when manually writing the config file, the user does not need to create the salt and digest pair but, instead, just specify a plaintext string to hash. The value will be properly saved as a salt/digest pair the next time the config file is saved to disk.

Available hash algorithms are:

- md5
- sha1
- sha224
- sha256
- sha384
- sha512

Parameters
hash_algorithm (str) – hash algorithm to use, must be a key of *ALGORITHMS*

ALGORITHMS: Dict[str, Callable[[bytes], hashlib._Hash]] = {'md5': <built-in function openssl_md5>, 'sha1': <built-in function openssl_sha1>, 'sha224': <built-in function openssl_sha224>, 'sha256': <built-in function openssl_sha256>, 'sha384': <built-in function openssl_sha384>, 'sha512': <built-in function openssl_sha512>}

Available hashing algorithms

storage_type

alias of *DigestValue*

to_basic(*cfg*, *value*)

Convert to a dict and indicate the type so we know on load whether we've already dealt with the field

Parameters

- **cfg** (*Config*) – current config
- **value** (*DigestValue*) – value to encrypt/hash

Return type
dict

Returns
encrypted/hashed value

to_python(*cfg*, *value*)

Decrypt the value if loading something we've already handled. Hash the value if it hasn't been hashed yet.

Parameters

- **cfg** (*Config*) – current config
- **value** (Union[dict, str]) – value to decrypt/load

Return type
DigestValue

Returns
decrypted value or unmodified hash

Raises
ValueError – if the value read from the config is neither a dict nor a string

class cincoconfig.**DigestValue**(*salt*, *digest*, *algorithm*)

Digest value tuple storing hashed value: (salt, digest, algorithm). The digest is the hash of the concatenated salt and plaintext value (`hash(salt + plaintext)`).

Create new instance of TDigestValue(salt, digest, algorithm)

challenge(*plaintext*)

Challenge a plaintext value against the digest value. This will raise a `ValueError` if the challenge is unsuccessful.

Raises
ValueError – the challenge was unsuccessful

Return type
None

classmethod create(*plaintext*, *algorithm*, *salt=None*)

Hash a plaintext value and return the new digest value. The digest is calculated as:

```
salt[:digest_size] + plaintext
```

The *salt* will be randomly generated if not specified. If the salt is specified and it is larger than the algorithm `digest_size`, the salt will be truncated to the `digest_size`.

Parameters

- **plaintext** – string to hash
- **algorithm** – hashlib algorithm to use
- **salt** – hash salt

Returns
the created digest value

classmethod parse(*value*, *algorithm*)

Parse a base64-encoded salt/digest pair, as returned by `__str__()`

class cincoconfig.**SecureField**(*method='best'*, *sensitive=True*, ***kwargs*)

A secure storage field where the plaintext configuration value is encrypted on disk and decrypted in memory when the configuration file is loaded.

Parameters
method (str) – encryption method, see *_get_provider()*

storage_type

alias of str

3.2 Instance Method Field

cincoconfig.**instance_method**(*schema*, *name*)

Bind a function to a schema as an instance method. Use this as a decorator:

```
schema = Schema()

@instance_method(schema, "say_hello")
def say_hello_method(config: Config) -> str:
```

(continues on next page)

(continued from previous page)

```
    return "Hello, world!"

config = schema()
print(config.say_hello())  # "Hello, world!"
```

Parameters

- **schema** (*Schema*) – schema
- **name** (str) – instance method name

Return type
Callable

3.3 Internal Types and Base fields

The following classes are used internally by cincoconfig and should not have to be used or referenced directly in applications. These are not included in the public API and must be imported explicitly from the `cincoconfig.fields` module.

class cincoconfig.**DictProxy**(*cfg*, *dict_field*, *iterable=None*)

A Field-validated dict proxy. This proxy supports all methods that the builtin `dict` supports with the added ability to validate keys and values against a *Field*. This is the value returned by the *DictField* validation chain.

property key_field: *Field*

Returns
the field for each item stored in the list.

property value_field: *Field*

Returns
the field for each item stored in the list.

class cincoconfig.**ListProxy**(*cfg*, *list_field*, *iterable=None*)

A Field-validated list proxy. This proxy supports all methods that the builtin `list` supports with the added ability to validate items against a *Field*. This is the field returned by the *ListField* validation chain.

property item_field: Union[BaseField, Type[*Config*]]

Returns
the field for each item stored in the list.

class cincoconfig.**NumberField**(*type_cls*, ***, *min=None*, *max=None*, ***kwargs*)

Base class for all number fields. This field should not be used directly, instead consider using *IntField* or *FloatField*.

Parameters

- **type_cls** (type) – number type class that values will be converted to
- **min** (Union[int, float, None]) – minimum value (inclusive)
- **max** (Union[int, float, None]) – maximum value (inclusive)

CHAPTER

FOUR

FORMATS

The following formats are included within cincoconfig. There should not be a reason to directly reference these classes. Instead, to save or load a configuration in the JSON format, for example, use:

```
config.save('config.json', format='json')
```

This will automatically create the *cincoconfig.formats.json.JsonConfigFormat*.

class cincoconfig.formats.**BsonConfigFormat**

BSON configuration file format. This format is only available when the `bson` package is installed.

This class should not be directly referenced. Instead, use the config *load()* and *save()* methods, passing *format='bson'*.

```
config.save('filename.bson', format='bson')
# or
config.load('filename.bson', format='bson')
```

dumps(*config*, *tree*)

Serialize the basic value `tree` to BSON `bytes` document.

Parameters

- **config** (*Config*) – current config
- **tree** (`dict`) – basic value tree

Return type
`bytes`

loads(*config*, *content*)

Deserialize the `content` (a `bytes` instance containing a BSON document) to a Python basic value tree.

Parameters

- **config** (*Config*) – current config
- **content** (`bytes`) – content to serialize

Return type
`dict`

Returns
the deserialized basic value tree

class cincoconfig.formats.**JsonConfigFormat**(*pretty=True*)

JSON configuration file format.

This class should not be directly referenced. Instead, use the config *load()* and *save()* methods, passing *format='json'*.

```
config.save('filename.json', format='json')
# or
config.load('filename.json', format='json')
```

Parameters
pretty (bool) – pretty-print the JSON document in the call to `json.dumps()`

dumps(*config*, *tree*)

Deserialize the `content` (a `bytes` instance containing a JSON document) to a Python basic value tree.

Parameters

- **config** (*Config*) – current config
- **content** – content to serialize

Return type
bytes

Returns
the deserialized basic value tree

loads(*config*, *content*)

Serialize the basic value `tree` to JSON `bytes` document.

Parameters

- **config** (*Config*) – current config
- **tree** – basic value tree

Return type
dict

class cincoconfig.formats.**PickleConfigFormat**

Python pickle configuration file format. This format uses the `pickle` module to serialize and deserialize the configuration basic value tree.

This class should not be directly referenced. Instead, use the config *load()* and *save()* methods, passing *format='pickle'*.

```
config.save('filename.cfg', format='pickle')
# or
config.load('filename.cfg', format='pickle')
```

dumps(*config*, *tree*)

Deserialize the `content` (a `bytes` instance containing a Pickled object) to a Python basic value tree.

Parameters

- **config** (*Config*) – current config
- **content** – content to serialize

Return type
bytes

Returns
the deserialized basic value tree

loads(*config*, *content*)

Serialize the basic value `tree` to PIckle `bytes` document.

Parameters

- **config** (*Config*) – current config
- **tree** – basic value tree

Return type
dict

class cincoconfig.formats.**XmlConfigFormat**(*root_tag='config'*)

XML configuration file format.

This class should not be directly referenced. Instead, use the config *load()* and *save()* methods, passing *format='xml'*.

```
config.save('filename.xml', format='xml')
# or
config.load('filename.xml', format='xml')
```

To handle dynamic configurations, the Python type for each XML element is stored in the `type` attribute.

```
<x type="int">1024</x>
```

When the configuration file is loaded, the formatter will attempt to parse the original Python type. If the parsing fails then the original string value is stored in the basic value tree.

Parameters
root_tag (str) – root configuration tag name

_to_element(*key*, *value*)

Convert the key/value pair to an XML element.

Parameters

- **key** (str) – the field key (becomes the tag name)
- **value** (Any) – the field valid

Return type
Element

Returns
the element containing the XML encoded key/value pair

_from_element(*ele*, *py_type=None*)

Parse the XML element to the original Python type. This method will attempt to convert any basic types to their original Python type and, if conversion fails, will use the original string value. For example:

```
<x type="int">blah</x>
```

This method will attempt to parse the value, *blah*, as a `int`, which will fail. Then, the method will store the original string value in the basic value tree:

```
tree = {
    'x': 'blah'
}
```

Parameters

- **ele** (Element) – the XML element to convert
- **py_type** (Optional[str]) – force the Python type attribute rather than reading the Python type from the *type* attribute

Return type
Any

Returns
the parsed Python value

_prettify(*ele*)

Pretty print the XML element.

Parameters
ele (Element) – XML element to pretty print

Return type
bytes

Returns
the pretty printed XML element

dumps(*config*, *tree*)

Serialize the basic value `tree` to an XML `bytes` document. The returned XML document will contain a single top-level tag named *root_key* that all other values are stored under.

Parameters

- **config** (*Config*) – current config
- **tree** (dict) – basic value tree

Return type
bytes

Returns
the serialized basic value tree

loads(*config*, *content*)

Deserialize the `content` (a `bytes` instance containing an XML document) to a Python basic value tree. The returned basic value tree will be scoped to *root_tag*, if it exists in the deserialized `dict`.

Parameters

- **config** (*Config*) – current config
- **content** (bytes) – content to deserialize

Return type
dict

Returns
deserialized basic value tree

class cincoconfig.formats.**YamlConfigFormat**(*root_key=None*)

YAML configuration file format. This format is only available when the `PyYAML` package is installed.

This class should not be directly referenced. Instead, use the config *load()* and *save()* methods, passing *format='yaml'*.

```
config.save('filename.yml', format='yaml')
# or
config.load('filename.yml', format='yaml')
```

By default, the basic value tree is serialized to YAML document as-is, where top-level configuration values are placed at the top level of the config file. For example:

```
>>> # assume tree = {'x': 1, 'y': 2}
>>> print(config.dumps(format='yaml'))
x: 1
y: 2
```

The *root_key* argument can be specified to store all configuration values under a single top-level key:

```
>>> # assume tree = {'x': 1, 'y': 2}
>>> print(config.dumps(format='yaml', root_key='CONFIG'))
CONFIG:
    x: 1
    y: 2
```

The *root_key* argument affects both how *loads()* and *dumps()* behave.

Parameters
root_key (Optional[str]) – the root config key that the configuration values should be stored under

dumps(*config*, *tree*)

Serialize the basic value `tree` to YAML `bytes` document. If *root_key* was specified, the returned YAML document will contain a single top-level field named *root_key* that all other values are stored under.

Parameters
- **config** (*Config*) – current config
- **tree** (dict) – basic value tree

Return type
bytes

Returns
the serialized basic value tree

loads(*config*, *content*)

Deserialize the `content` (a `bytes` instance containing a YAML document) to a Python basic value tree. If *root_key* was specified, the returned basic value tree will be scoped to *root_key*, if it exists in the deserialized `dict`. This is equivalent to:

```
tree = yaml.load(content)
return tree[self.root_key]
```

Parameters
- **config** (*Config*) – current config

- **content** (bytes) – content to deserialize

Return type
dict

Returns
deserialized basic value tree

4.1 Base Classes

All configuration file formats must inherit from and implement all abstract methods in the `cincoconfig.abc.ConfigFormat` class.

class cincoconfig.**ConfigFormat**

The base class for all configuration file formats.

dumps(*config*, *tree*)

Convert the configuration value tree to a bytes object. This method is called to serialize the configuration to a buffer and eventually write to a file.

Parameters

- **config** (*Config*) – current configuration
- **tree** (dict) – basic value tree, as returned by the Config `to_tree()` method.

Return type
bytes

Returns
the serialized configuration

classmethod get(*name*, ***kwargs*)

Get a registered configuration format.

Parameters

- **name** (str) – config format name
- **kwargs** – keyword arguments to pass into the config format `__init__()` method

Return type
ConfigFormat

Returns
the config format instance

classmethod initialize_registry()

Initialize the format registry for built-in formats.

Return type
None

loads(*config*, *content*)

Parse the serialized configuration to a basic value tree that can be parsed by the Config `load_tree()` method.

Parameters

- **config** (*Config*) – current config
- **content** (bytes) – serialized content

Return type
dict

Returns
the parsed basic value tree

classmethod register(*name*, *format_cls*)

Register a new configuration format.

Parameters

- **name** (str) – format name
- **format_cls** (Type[*ConfigFormat*]) – ConfigFormat subclass to register

Return type
None

CHAPTER

FIVE

ENCRYPTION

class cincoconfig.**KeyFile**(*filename*)

The cincoconfig key file, containing a randomly generated 32 byte encryption key. The cinco key file is used by *SecureField* to encrypt and decrypt values as they are written to and read from the configuration file.

The keyfile is loaded as needed by using this class as a context manager. The key has an internal reference count and is only freed once the reference count is 0 (all context managers exited). THe key is cached internally so that the keyfile only has to be open and read once per configuration load or save.

To encrypt a value:

```
with keyfile as ctx:
    secret = ctx.encrypt(method='xor', text='hello, world')
```

Parameters
filename (str) – the cinco key filename

_get_provider(*method*)

Get the encryption provider. `method` must be one of

- `aes` - returns `AesProvider`
- `xor` - returns `XorProvider`
- `best` - returns the best available encryption provider: `AesProvider` if AES encryption is available (`cryptography` is installed), `XorProvider` if AES is not available

The resolved method is returned. For example, if `best` if specified, the best encryption method will be resolved and returned.

The return value is a tuple of encryption provider instance and the resolved method.

Return type
Tuple[*IEncryptionProvider*, str]

Returns
a tuple of `(provider, method)`

decrypt(*secret*)

Parameters
secret (*SecureValue*) – encrypted value

Return type
bytes

Returns
decrypted value

encrypt(*text*, *method='best'*)

Parameters

- **text** (Union[str, bytes]) – plaintext to encrypt
- **method** (str) – encryption method to use

Return type
SecureValue

Returns
the encrypted value

generate_key()

Generate a random 32 byte key and save it to `filename`.

Return type
None

class cincoconfig.encryption.**SecureValue**(*method*, *ciphertext*)

An encrypted value tuple containing the encryption method and the ciphertext.

method

the encryption method (str)

ciphertext

the encrypted value (bytes)

cincoconfig.encryption.**AES_AVAILABLE**

AES is available (`cryptography` is installed)

5.1 Internal Classes

class cincoconfig.encryption.**IEncryptionProvider**

Interface class for an encryption algorithm provider. An encryption provider implements both encryption and decryption of string values.

The encrypt and decrypt methods must be deterministic.

```
b'message' == provider.decrypt(provider.decrypt(b'message'))
```

The constructor for subclasses will receive a single argument: the encryption key.

decrypt(*ciphertext*)

Decrypt a value.

Parameters
ciphertext (bytes) – encrypted value to decrypt

Return type
bytes

Returns
decrypted value

encrypt(*text*)

Encrypt a value.

Parameters
text (bytes) – plain text value to encrypt

Return type
bytes

Returns
encrypted value

class cincoconfig.encryption.**XorProvider**(*key*)

XOR-bitwise "encryption". The XOR provider should only be used to obfuscate, not encrypt, a value since XOR operations can be easily reversed.

decrypt(*ciphertext*)

Return type
bytes

Returns
the decrypted values

encrypt(*text*)

Return type
bytes

Returns
the encrypted value

class cincoconfig.encryption.**AesProvider**(*key*)

AES-256 encryption provider. This class requires the `cryptography` library. Each encrypted value has a randomly generated 16-byte IV.

decrypt(*ciphertext*)

Return type
bytes

Returns
the plaintext value

encrypt(*text*)

Return type
bytes

Returns
the encrypted value

CHAPTER

SIX

STUBS

Cincoconfig configuration objects are dynamic in nature which makes working with a cincoconfig object cumbersome inside of an IDE that attempts to provide autocomplete / Intellisense results. To work around this, Python type stub files, `*.pyi`, can be used that define the structure of a cincoconfig configuration so that the IDE or type checker can work properly with configuration objects.

cincoconfig.**generate_stub**(*config*, *class_name=None*)

Generate the Python stub class (pyi file) for a provided Schema instance, Config instance, or ConfigType class. Generating a pyi stub file is useful when developing in an IDE, such as VSCode, to make the autocompleter / Intellisense / Language Server understand the structure of the configuration and properly show autocomplete results and perform type checking / linting.

Save the generated pyi stub file to a location in your project repository and then configure the IDE / type checked (MyPy) to use the directory to load additional type information from.

Parameters

- **config** (Union[*Schema*, *ConfigType*, *Config*]) – the configuration to generate the stub file from
- **class_name** (Optional[str]) – the configuration class name

Return type
str

Returns
the content of the pyi stub file for the provided configuration

CHAPTER

SEVEN

SUPPORT

cincoconfig.**make_type**(*schema*, *name*, *module=None*, *key_filename=None*)

Create a new type that wraps this schema. This method should only be called once per schema object.

Use this method when to create reusable configuration objects that can be used multiple times in code in a more traditional Pythonic manner. For example, consider the following:

```
item_schema = Schema()
item_schema.url = UrlField()
item_schema.verify_ssl = BoolField(default=True)

schema = Schema()
schema.endpoints = ListField(item_schema)

config = schema()

# to create new web hook items
item = webhook_schema()
item.url = 'https://google.com'
item.verify_ssl = False

config.endpoints.append(item)
```

This is a cumbersome design when creating these objects within code. `make_type` will dynamically create a new class that can be used in a more Pythonic way:

```
# same schema as above
config = schema()
Item = make_type(item_schema, 'Item')

item = Item(url='https://google.com', verify_ssl=False)
config.endpoints.append(item)
```

The new class inherits from *Config*.

Parameters

- **name** (str) – the new class name
- **module** (Optional[str]) – the owning module
- **key_filename** (Optional[str]) – the key file name passed to each new config object,
- **validator** – config validator callback method

Return type
Type[*ConfigType*]

Returns
the new type

cincoconfig.**generate_argparse_parser**(*schema*, ***parser_kwargs*)

Generate a argparse.ArgumentParser based on the schema. This method generates `--long-arguments` for each field that stores a string, integer, float, or bool (based on the field's `storage_type`). Boolean fields have two long arguments created, one to store a `True` value and another, `--no-[x]`, to disable it.

Parameters
kwargs – keyword arguments to pass to the generated `ArgumentParser` constructor

Return type
ArgumentParser

Returns
the generated argument parser

cincoconfig.**validator**(*field*)

Decorator to register a new validator with the schema or field. All validators will be run against the configuration whenever the configuration is loaded from disk. Multiple validators can be registered by using the decorator multiple times. Subconfigs can also be validated by using the decorator on the sub schema.

```
schema = Schema()
schema.x = IntField()
schema.y = IntField()
schema.db.username = StringField()
schema.db.password = StringField()

@validator(schema)
def validate_x_lt_y(cfg):
    if cfg.x and cfg.y and cfg.x >= cfg.y:
        raise ValueError('x must be less-than y')

@validator(schema.db)
def validate_db_credentials(cfg):
    if cfg.username and not db.password:
        raise ValueError('db.password is required when username is specified')

config = schema()
config.load('config.json', format='json')  # will call the above validators
# .....
```

The validator function needs to raise an exception, preferably a ValueError, if the validation fails.

Parameters
func – validator function that accepts a single argument: *Config*.

Return type
Callable

Returns
func

cincoconfig.**get_all_fields**(*schema*)

Get all the fields and nested fields of the schema or config, including the nested schemas/configs.

```
>>> schema = Schema()
>>> schema.x = IntField()
>>> schema.y.z = StringField()
>>> schema.z = StringField()
>>> get_all_fields(schema)
[
    ('x', schema, schema.x),
    ('y.z', schema.y, schema.y.z),
    ('z', schema, schema.z)
]
```

The returned list of tuples have three values:

1. *path* - the full path to the field.
2. *schema* - the schema that the field belongs to.
3. *field* - the field.

The order of the fields will be the same order in which the fields were added to the schema.

Return type
List[Tuple[str, *Schema*, BaseField]]

Returns
all the fields as a list of tuples: `(path, schema, field)`

cincoconfig.**cmdline_args_override**(*config*, *args*, *ignore=None*)

Override configuration setting based on command line arguments, parsed from the `argparse` module. This method is useful when loading a configuration but allowing the user the option to override or extend the configuration via command line arguments.

```
parser = argparse.ArgumentParser()
parser.add_argument('-d', '--debug', action='store_const', const='debug', dest='mode
↪')
parser.add_argument('-p', '--port', action='store', dest='http.port')
parser.add_argument('-H', '--host', action='store', dest='http.address')
parser.add_argument('-c', '--config', action='store')

args = parser.parse_args()
if args.config:
    config.load(args.config, format='json')

config.cmdline_args_override(args, ignore=['config'])

# cmdline_args_override() is equivalent to doing:

if args.mode:
    config.mode = args.mode
if getattr(args, 'http.port'):
    config.http.port = getattr(args, 'http.port')
if getattr(args, 'http.address'):
    config.http.address = getattr(args, 'http.address')
```

This method is compatible with manually created argument parser and an autogenerated one from the Schema *generate_argparse_parser()* method.

Parameters

- **args** (Namespace) – parsed command line arguments from parse_args()
- **ignore** (Union[str, List[str], None]) – list of arguments to ignore and not process

Return type
None

cincoconfig.**item_ref_path**(*item*)

Get the full reference path to a field or configuration.

Parameters
item (Union[BaseField, *Config*]) – field, schema, or configuration

Return type
str

Returns
full reference path to the item

cincoconfig.**get_fields**(*schema*, *types=None*)

Get all fields within a configuration or schema. This method does not recurse into nested schemas/configs, unlike *get_all_fields()*. The return value is a list of fields as tuples `(key, field)`.

Parameters

- **schema** (Union[*Config*, *Schema*]) – schema or configuration object
- **types** (Union[Type, Tuple[Type], None]) – only return fields of a specific type

Return type
List[Tuple[str, BaseField]]

Returns
the list of fields

cincoconfig.**asdict**(*config*, *virtual=False*)

Converts the configuration object to a dict. Whereas *to_tree()* converts the configuration to a basic value tree suitable for saving to disk, the `asdict` method converts the configuration, and all nested configuration objects, to a dict, preserving each value as-is.

Parameters

- **config** (*Config*) – the configuration to convert to a dict
- **virtual** (bool) – include virtual field values in the dict

Return type
dict

Returns
the converted configuration dict

cincoconfig.**is_value_defined**(*config*, *key*)

Check if the given field has been set by the user through either loading a configuration file or using the API to set the field value.

Parameters

- **config** (*Config*) – configuration object
- **key** (str) – field key

Return type
bool

Returns
the field is set by the user

cincoconfig.**reset_value**(*config*, *key*)

Reset a config value back to the default.

Parameters

- **config** (*Config*) – configuration object
- **key** (str) – field key

Return type
None

cincoconfig is an easy to use configuration file management library. It allows you to build custom application and library configurations declaratively without any subclassing or specializations. Cincoconfig ships with 20+ builtin *fields* with comprehensive value validation.

cincoconfig has no hard dependencies, however, several features become available by installing several dependencies, including (see *requirements/requirements-features.txt* for full list):

- Configuration value encryption (*SecureField*) - requires cryptography
- YAML config file support - requires PyYaml
- BSON config file support - requires bson

Configuration values are directly accessed as attributes.

```
# app_config.py
from cincoconfig import *

# first, define the configuration's schema -- the fields available that
# customize the application's or library's behavior
schema = Schema()
schema.mode = ApplicationModeField(default='production')

# nested configurations are built on the fly
# http is now a subconfig
schema.http.port = PortField(default=8080, required=True)

# each field has its own validation rules that are run anytime the config
# value is loaded from disk or modified by the user.
# here, this field only accepts IPv4 network addresses and the user is
# required to define this field in the configuration file.
schema.http.address = IPv4AddressField(default='127.0.0.1', required=True)

schema.http.ssl.enabled = BoolField(default=False)
schema.http.ssl.ca_file = FilenameField()
schema.http.ssl.key_file = FilenameField()
schema.http.ssl.cert_file = FilenameField()

schema.db.host = HostnameField(allow_ipv4=True, required=True, default='localhost')
schema.db.port = PortField(default=27017, required=True)
schema.db.name = StringField(default='my_app', required=True)
```

(continues on next page)

(continued from previous page)

```
schema.db.user = StringField()

# some configuration values are sensitive, such as credentials, so
# cincoconfig provides config value encryption when the value is
# saved to disk via the SecureField
schema.db.password = SecureField()


# once a schema is defined, build the actual configuration object
# that can load config files from disk and interact with the values
config = schema()

# print the set http port
print(config.http.port) # >>> 8080

# set a config value manually
if config.mode == 'production':
    config.db.name = config.db.name + '_production'

print(config.dumps(format='json', pretty=True))
# {
#   "mode": "production",
#   "http": {
#     "port": 8080,
#     "address": "127.0.0.1"
#     "ssl": {
#       "enabled": false
#     }
#   },
#   "db": {
#     "host": "localhost",
#     "port": 27017,
#     "name": "my_app_production"
#   }
# }
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

T

U

V

X

Y