lazr.config

Release 3.0

unknown

Jun 20, 2023

CONTENTS

The LAZR config system is typically used to manage process configuration. Process configuration is for saying how things change when we run systems on different machines, or under different circumstances.

This system uses ini-like file format of section, keys, and values. The config file supports inheritance to minimize duplication of information across files. The format supports schema validation.

CHAPTER

ONE

CONFIGSCHEMA

A schema is loaded by instantiating the ConfigSchema class with the path to a configuration file. The schema is explicitly derived from the information in the configuration file.

```
>>> from pkg_resources import resource_string
>>> raw_schema = resource_string('lazr.config.tests.testdata', 'base.conf')
```

The config file contains sections enclosed in square brackets (e.g. `[section]`). The section name may be divided into major and minor categories using a dot (`.`). Beneath each section is a list of key-value pairs, separated by a colon (`:`).

Multiple sections with the same major category may have their keys defined in another section that appends the `.template` suffix to the category name.

A section with `.optional` suffix is not required. Lines that start with a hash (`#`) are comments.

```
>>> from pkg_resources import resource_string
>>> raw_schema = resource_string('lazr.config.tests.testdata', 'base.conf')
>>> print(raw_schema.decode('utf-8'))
# This section defines required keys and default values.
[section_1]
key1: foo
key2: bar and baz
key3: Launchpad rocks
key4: F&#028c;k yeah!
key5:
# This section is required, and it defines all the keys for its category.
[section-2.app-b]
key1: True
# This section is optional; it uses the keys defined
# by section_3.template.
[section_3.app_a.optional]
# This is a required section whose keys are defined by section_3.template
# and it defines a new key.
[section_3.app_b]
key2: changed
key3: unique
# These sections define a common set of required keys and default values.
[section_3.template]
key1: 17
key2: 3.1415
# This section is optional.
[section-5.optional]
key1: something
```

(continues on next page)

(continued from previous page)

```
# This section has a name similar to a category.
[section_33]
key1: fnord
key2: multiline value 1
   multiline value 2
```

To create the schema, provide a file name.

```
>>> from lazr.config import ConfigSchema
>>> from lazr.config.interfaces import IConfigSchema
>>> from pkg_resources import resource_filename
>>> from zope.interface.verify import verifyObject
>>> base_conf = resource_filename(
...     'lazr.config.tests.testdata', 'base.conf')
>>> schema = ConfigSchema(base_conf)
>>> verifyObject(IConfigSchema, schema)
True
```

The schema has a name and a file name.

```
>>> print(schema.name)
base.conf
>>> print('file:', schema.filename)
file: ...lazr/config/tests/testdata/base.conf
```

If you provide an optional file-like object as a second argument to the constructor, that is used instead of opening the named file implicitly.

```
>>> with open(base_conf, 'r') as file_object:
...     other_schema = ConfigSchema('/does/not/exist.conf', file_object)
>>> verifyObject(IConfigSchema, other_schema)
True
```

For such schemas, the file name is taken from the first argument.

```
>>> print(other_schema.name)
exist.conf
>>> print(other_schema.filename)
/does/not/exist.conf
```

A schema is made up of multiple SchemaSections. They can be iterated over in a loop as needed.

```
>>> from operator import attrgetter
>>> for section_schema in sorted(schema, key=attrgetter('name')):
...     print(section_schema.name)
section-2.app-b
section-5
section_1
section_3.app_a
section_3.app_b
section_33
```

```
>>> for section_schema in sorted(other_schema, key=attrgetter('name')):
...     print(section_schema.name)
section-2.app-b
section-5
section_1
section_3.app_a
section_3.app_b
section_33
```

You can check if the schema contains a section name, and that can be used to access the SchemaSection as a subscript.

```
>>> 'section_1' in schema
True
>>> 'section-4' in schema
False
```

A SectionSchema can be retrieved from the schema using the `[]` operator.

```
>>> section_schema_1 = schema['section_1']
>>> print(section_schema_1.name)
section_1
```

Processes often require resources like databases or virtual hosts that have a common category of keys. The list of all category names can be retrieved via the categories attribute.

```
>>> for name in schema.category_names:
...     print(name)
section-2
section_3
```

The list of SchemaSections that share common category can be retrieved using `getByCategory()`.

```
>>> all_section_3 = schema.getByCategory('section_3')
>>> for section_schema in sorted(all_section_3, key=attrgetter('name')):
...     print(section_schema.name)
section_3.app_a
section_3.app_b
```

You can pass a default argument to `getByCategory()` to avoid the exception.

```
>>> missing = object()
>>> schema.getByCategory('non-section', missing) is missing
True
```

CHAPTER

TWO

SCHEMASECTION

A SchemaSection behaves similar to a dictionary. It has keys and values.

```
>>> from lazr.config.interfaces import ISectionSchema
>>> section_schema_1 = schema['section_1']
>>> verifyObject(ISectionSchema, section_schema_1)
True
```

Each SchemaSection has a name.

```
>>> print(section_schema_1.name)
section_1
```

A SchemaSection can return a 2-tuple of its category name and specific name parts.

```
>>> for name in schema['section_3.app_b'].category_and_section_names:
...     print(name)
section_3
app_b
```

The category name will be `None` if the SchemaSection's name does not contain a category.

```
>>> for name in section_schema_1.category_and_section_names:
...     print(name)
None
section_1
```

Optional sections have the optional attribute set to `True`:

```
>>> section_schema_1.optional
False
>>> schema['section_3.app_a'].optional
True
```

A key can be verified to be in a section.

```
>>> 'key1' in section_schema_1
True
>>> 'nonkey' in section_schema_1
False
```

A key can be accessed directly using as a subscript of the SchemaSection. The value is always a string.

```
>>> print(section_schema_1['key3'])
Launchpad rocks
>>> section_schema_1['key5']
''
```

An error is raised if a non-existent key is accessed.

```
>>> section_schema_1['not-exist']
Traceback (most recent call last):
  ...
KeyError: ...
```

In the conf file, `[section_1]` is a default section that defines keys and values. The values specified in the section schema will be used as default values if not overridden in the configuration. In the case of *key5*, the key had no explicit value, so the value is an empty string.

```
>>> for key in sorted(section_schema_1):
...     print(key, ':', section_schema_1[key])
key1 : foo
key2 : bar and baz
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 :
```

In the conf file `[section_3.template]` defines a common set of keys and default values for `[section_3.app_a]` and `[section_3.app_b]`. When a section defines different keys and default values from the template, the new data overlays the template data. This is the case for section `[section_3.app_b]`.

```
>>> for section_schema in sorted(all_section_3, key=attrgetter('name')):
...     print(section_schema.name)
...     for key in sorted(section_schema):
...         print(key, ':', section_schema[key])
section_3.app_a
key1 : 17
key2 : 3.1415
section_3.app_b
key1 : 17
key2 : changed
key3 : unique
```

CHAPTER

THREE

CONFIGSCHEMA VALIDATION

The schema parser is self-validating. It checks that the character encoding is ASCII, and that the data is not ambiguous or self-contradicting. Keys must exist inside sections and section names may not be defined twice. Sections may belong to only one category, and only letters, numbers, dots and dashes may be present in section names.

CHAPTER

FOUR

ICONFIGLOADER

ConfigSchema implements the two methods in the IConfigLoader interface. A Config is created by a schema using either the `load()` or `loadFile()` methods to return a Config instance.

```
>>> from lazr.config.interfaces import IConfigLoader
>>> verifyObject(IConfigLoader, schema)
True
```

The `load()` method accepts a filename.

```
>>> local_conf = resource_filename(
...     'lazr.config.tests.testdata', 'local.conf')
>>> config = schema.load(local_conf)
```

The `loadFile()` method accepts a file-like object and an optional filename keyword argument. The filename argument must be passed if the file-like object does not have a `name` attribute.

```
>>> try:
...     from io import StringIO
... except ImportError:
...     # Python 2
...     from StringIO import StringIO
>>> bad_data = ("""
...     [meta]
...     metakey: unsupported
...     [unknown-section]
...     key1 = value1
...     [section_1]
...     keyn: unknown key
...     key1: bad character in caf\xc3)
...     [section_3.template]
...     key1: schema suffixes are not permitted""")
>>> bad_config = schema.loadFile(
...     StringIO(bad_data), 'bad conf')
```

CHAPTER

FIVE

CONFIG

The config represents the local configuration of the process on a system. It is validated with a schema. It extends the schema, or other conf files, to define the specific differences from the extended files that are required to run the local processes.

The object returned by `load()` provides both the `IConfigData` and `IStackableConfig` interfaces. `IConfigData` is for read-only access to the configuration data. A process configuration is made up of a stack of different `IConfigData`. The `IStackableConfig` interface provides the methods used to manipulate that stack of configuration overlays.

```
>>> from lazr.config.interfaces import IConfigData, IStackableConfig
>>> verifyObject(IConfigData, config)
True
>>> verifyObject(IStackableConfig, config)
True
```

Like the schema file, the conf file is made up of sections with keys. The sections may belong to a category. Unlike the schema file, it does not have template or optional sections. The `[meta]` section has the extends key that declares that this conf extends `shared.conf`.

```
>>> with open(local_conf, 'rt') as local_file:
...     raw_conf = local_file.read()
>>> print(raw_conf)
[meta]
extends: shared.conf
# Localize a key for section_1.
[section_1]
key5: local value
# Accept the default values for the optional section-5.
[section-5]
```

The `.master` section allows admins to define configurations for an arbitrary number of processes. If the schema defines `.master` sections, then the conf file can contain sections that extend the `.master` section. These are like categories with templates except that the section names extending `.master` need not be named in the schema file.

```
>>> master_schema_conf = resource_filename(
...     'lazr.config.tests.testdata', 'master.conf')
>>> master_local_conf = resource_filename(
...     'lazr.config.tests.testdata', 'master-local.conf')
>>> master_schema = ConfigSchema(master_schema_conf)
>>> sections = master_schema.getByCategory('thing')
>>> for name in sorted(section.name for section in sections):
...     print(name)
```

(continues on next page)

(continued from previous page)

```
thing.master
>>> master_conf = master_schema.load(master_local_conf)
>>> sections = master_conf.getByCategory('thing')
>>> for name in sorted(section.name for section in sections):
...     print(name)
thing.one
thing.two
>>> for name in sorted(section.foo for section in sections):
...     print(name)
1
2
>>> print(master_conf.thing.one.name)
thing.one
```

The `shared.conf` file derives the keys and default values from the schema. This config was loaded before `local.conf` because its sections and values are required to be in place before `local.conf` applies its changes.

```
>>> shared_config = resource_filename(
...     'lazr.config.tests.testdata', 'shared.conf')
>>> with open(shared_config, 'rt') as shared_file:
...     raw_conf = shared_file.read()
>>> print(raw_conf)
# The schema is defined by base.conf.
# Localize a key for section_1.
[section_1]
key2: sharing is fun
key5: shared value
```

The config that was loaded has `name` and `filename` attributes to identify the configuration.

```
>>> print(config.name)
local.conf
>>> print('file:', config.filename)
file: ...lazr/config/tests/testdata/local.conf
```

The config can access the schema via the schema property.

```
>>> print(config.schema.name)
base.conf
>>> config.schema is schema
True
```

A config is made up of multiple Sections like the schema. They can be iterated over in a loop as needed. This config inherited several sections defined in schema. Note that the meta section is not present because it pertains to the config system, not to the processes being configured.

```
>>> for section in sorted(config, key=attrgetter('name')):
...     print(section.name)
section-2.app-b
section-5
section_1
section_3.app_b
section_33
```

You can check if a section name is in a config.

```
>>> 'section_1' in config
True
>>> 'bad-section' in config
False
```

Optional SchemaSections are not inherited by the config. A config file must declare all optional sections. Including the section heading is enough to inherit the section and its keys. The config file may localize the keys by declaring them too. The `local.conf` file includes `section-5`, but not `section_3.app_a`.

```
>>> 'section_3.app_a' in config
False
>>> 'section_3.app_a' in config.schema
True
>>> config.schema['section_3.app_a'].optional
True
>>> 'section-5' in config
True
>>> 'section-5' in config.schema
True
>>> config.schema['section-5'].optional
True
```

A Section can be accessed using subscript notation. Accessing a section that does not exist will raise a NoSectionError. NoSectionError is raised for a undeclared optional sections too.

```
>>> section_1 = config['section_1']
>>> section_1.name in config
True
```

Config supports category access like Schema does. The list of categories are returned by the `category_names` property.

```
>>> for name in sorted(config.category_names):
...     print(name)
section-2
section_3
```

All the sections that belong to a category can be retrieved using the `getByCategory()` method.

```
>>> for section in config.getByCategory('section_3'):
...     print(section_schema.name)
section_3.app_b
```

Passing a non-existent category_name to the method will raise a NoCategoryError. As with schemas, you can pass a default argument to `getByCategory()` to avoid the exception.

```
>>> missing = object()
>>> config.getByCategory('non-section', missing) is missing
True
```

CHAPTER

SIX

SECTION

A Section behaves similar to a dictionary. It has keys and values. It supports some specialize access methods and properties for working with the values. Each Section has a name.

```
>>> from lazr.config.interfaces import ISection
>>> verifyObject(ISection, section_1)
True
>>> print(section_1.name)
section_1
```

Like SectionSchemas, sections can return a 2-tuple of their category name and specific name parts. The category name will be `None` if the section's name does not contain a category.

```
>>> for name in config['section_3.app_b'].category_and_section_names:
...     print(name)
section_3
app_b
>>> for name in section_1.category_and_section_names:
...     print(name)
None
section_1
```

The Section's type is the same type as the `ConfigSchema.section_factory`.

```
>>> section_1
<lazr.config...Section object at ...>
>>> config.schema.section_factory
<class 'lazr.config...Section'>
```

A key can be verified to be in a Section.

```
>>> 'key1' in section_1
True
>>> 'nonkey' in section_1
False
```

A key can be accessed directly using as a subscript of the Section. The value is always a string.

```
>>> print(section_1['key3'])
Launchpad rocks
>>> print(section_1['key5'])
local value
```

An error is raised if a non-existent key is accessed via a subscript.

```
>>> section_1['not-exist']
Traceback (most recent call last):
  ...
KeyError: ...
```

The Section keys can be iterated over. The section has all the keys from the SectionSchema. The values came form the schema's default values, then the values from `shared.conf` were applied, and lastly, the values from `local.conf` were applied. The schema provided the values of `key1`, `key3`, and `key4`. `shared.conf` provided the value of `key2` . `local.conf` provided `key5`. While `shared.conf` provided a `key5`, `local.conf` takes precedence.

```
>>> for key in sorted(section_1):
...     print(key, ':', section_1[key])
key1 : foo
key2 : sharing is fun
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 : local value
>>> section_1.schema['key5']
''
```

The schema provided mandatory sections and default values to the config. So while the config file did not declare all the sections, they are present. In the case of `section_3.app_b`, its keys were defined in a template section.

```
>>> for key in sorted(config['section_3.app_b']):
...     print(key, ':', config['section_3.app_b'][key])
key1 : 17
key2 : changed
key3 : unique
```

Sections attributes cannot be directly set to shadow config options. An `AttributeError` is raised when an attempt is made to mutate the config.

```
>>> config['section_3.app_b'].key1 = 'fail'
Traceback (most recent call last):
 ...
AttributeError: Config options cannot be set directly.
```

Nor can new attributes be added to a section.

```
>>> config['section_3.app_b'].no_such_attribute = 'fail'
Traceback (most recent call last):
 ...
AttributeError: Config options cannot be set directly.
```

CHAPTER

SEVEN

VALIDATING CONFIGS

Config provides the `validate()` method to verify that the config is valid according to the schema. The method returns `True` if the config is valid.

```
>>> config.validate()
True
```

When the config is not valid, a ConfigErrors is raised. The exception has an `errors` property that contains a list of all the errors in the config.

CHAPTER

EIGHT

CONFIG OVERLAYS

A conf file may contain a meta section that is used by the config system. The config data can access the config it extended using the `extends` property. The object is just the config data; it does not have any config methods.

```
>>> print(config.extends.name)
shared.conf
```

```
>>> verifyObject(IConfigData, config.extends)
True
```

As Config supports inheritance through the `extends` key, each conf file produces instance of ConfigData, called an *overlay*. ConfigData represents the state of a config. The `overlays` property is a stack of ConfigData as it was constructed from the schema's config to the last config file that was loaded.

```
>>> for config_data in config.overlays:
...     print(config_data.name)
local.conf
shared.conf
base.conf
>>> verifyObject(IConfigData, config.overlays[-1])
True
```

Conf files can use the `extends` key to specify that it extends a schema without incurring a processing penalty by loading the schema twice in a row. The schema can never be the second item in the overlays stack.

```
>>> single_config = schema.load(schema.filename)
>>> for config_data in single_config.overlays:
...     print(config_data.name)
base.conf
>>> single_config.push(schema.filename, raw_schema.decode('utf-8'))
>>> for config_data in single_config.overlays:
...     print(config_data.name)
base.conf
```

CHAPTER

NINE

PUSH()

Raw config data can be merged with the config to create a new overlay for testing. The `push()` method accepts a string of config data. The data must conform to the schema. The `section_1` sections's keys are updated when the unparsed data is pushed onto the config. Note that indented, unparsed data is passed to `push()` in this example; `push()` does not require tests to dedent the test data.

```
>>> for key in sorted(config['section_1']):
...     print(key, ':', config['section_1'][key])
key1 : foo
key2 : sharing is fun
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 : local value

>>> test_data = ("""
...     [section_1]
...     key1: test1
...     key5:""")
>>> config.push('test config', test_data)

>>> for key in sorted(config['section_1']):
...     print(key, ':', config['section_1'][key])
key1 : test1
key2 : sharing is fun
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 :
```

Besides updating section keys, optional sections can be enabled too. The `section_3.app_a` section is enabled with the default keys from the schema in this example.

```
>>> config.schema['section_3.app_a'].optional
True
>>> 'section_3.app_a' in config
False

>>> app_a_data = "[section_3.app_a]"
>>> config.push('test app_a', app_a_data)

>>> 'section_3.app_a' in config
True
>>> for key in sorted(config['section_3.app_a']):
```

(continues on next page)

(continued from previous page)

```
...     print(key, ':', config['section_3.app_a'][key])
key1 : 17
key2 : 3.1415

>>> for key in sorted(config.schema['section_3.app_a']):
...     print(key, ':', config.schema['section_3.app_a'][key])
key1 : 17
key2 : 3.1415
```

The config's name and overlays are updated by `push()`.

```
>>> print(config.name)
test app_a
>>> print(config.filename)
test app_a
>>> for config_data in config.overlays:
...     print(config_data.name)
test app_a
test config
local.conf
shared.conf
base.conf
```

The `test app_a` config did not declare an `extends` key in a `meta` section. Its `extends` property is `None`, even though it implicitly extends `test config`. The `extends` property only provides access to configs that are explicitly extended.

```
>>> print(config.extends.name)
test config
```

The config's sections are updated with `section_3.app_a` too.

```
>>> for section in sorted(config, key=attrgetter('name')):
...     print(section.name)
section-2.app-b
section-5
section_1
section_3.app_a
section_3.app_b
section_33
```

A config file may state that it extends its schema (to clearly connect the config to the schema). The schema can also be pushed to reset the values in the config to the schema's default values.

```
>>> extender_conf_name = resource_filename(
...     'lazr.config.tests.testdata', 'extender.conf')
>>> extender_conf_data = ("""
...     [meta]
...     extends: base.conf""")
>>> config.push(extender_conf_name, extender_conf_data)
>>> for config_data in config.overlays:
...     print(config_data.name)
extender.conf
base.conf
```

(continues on next page)

(continued from previous page)

```
test app_a
test config
local.conf
shared.conf
base.conf
```

The section_1 section was restored to the schema's default values.

```
>>> for key in sorted(config['section_1']):
...     print(key, ':', config['section_1'][key])
key1 : foo
key2 : bar and baz
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 :
```

push() can also be used to extend master sections.

```
>>> sections = sorted(master_conf.getByCategory('bar'),
...                   key=attrgetter('name'))
>>> for section in sections:
...     print(section.name, section.baz)
bar.master badger
bar.soup cougar

>>> master_conf.push('override', """
... [bar.two]
... baz: dolphin
... """)
>>> sections = sorted(master_conf.getByCategory('bar'),
...                   key=attrgetter('name'))
>>> for section in sections:
...     print(section.name, section.baz)
bar.soup cougar
bar.two dolphin

>>> master_conf.push('overlord', """
... [bar.three]
... baz: emu
... """)
>>> sections = sorted(master_conf.getByCategory('bar'),
...                   key=attrgetter('name'))
>>> for section in sections:
...     print(section.name, section.baz)
bar.soup cougar
bar.three emu
bar.two dolphin
```

push() works with master sections too.

```
>>> schema_file = StringIO("""\
... [thing.master]
... foo: 0
```

(continues on next page)

(continued from previous page)

```
... bar: 0
... """)
>>> push_schema = ConfigSchema('schema.cfg', schema_file)

>>> config_file = StringIO("""\
... [thing.one]
... foo: 1
... """)
>>> push_config = push_schema.loadFile(config_file, 'config.cfg')
>>> print(push_config.thing.one.foo)
1
>>> print(push_config.thing.one.bar)
0

>>> push_config.push('test.cfg', """\
... [thing.one]
... bar: 2
... """)
>>> print(push_config.thing.one.foo)
1
>>> print(push_config.thing.one.bar)
2
```

CHAPTER

TEN

POP()

ConfigData can be removed from the stack of overlays using the `pop()` method. The methods returns the list of ConfigData that was removed – a slice from the specified ConfigData to the top of the stack.

```
>>> overlays = config.pop('test config')
>>> for config_data in overlays:
...     print(config_data.name)
extender.conf
base.conf
test app_a
test config

>>> for config_data in config.overlays:
...     print(config_data.name)
local.conf
shared.conf
base.conf
```

The config's state was restored to the ConfigData that is on top of the overlay stack. Section `section_3.app_a` was removed completely. The keys (`key1` and `key5`) for `section_1` were restored.

```
>>> for section in sorted(config, key=attrgetter('name')):
...     print(section.name)
section-2.app-b
section-5
section_1
section_3.app_b
section_33

>>> for key in sorted(config['section_1']):
...     print(key, ':', config['section_1'][key])
key1 : foo
key2 : sharing is fun
key3 : Launchpad rocks
key4 : F&#028c;k yeah!
key5 : local value
```

A Config must have at least one ConfigData in the overlays stack so that it has data. The bottom ConfigData in the overlays was made from the schema's required sections. It cannot be removed by the `pop()` method.

If all but the bottom ConfigData is popped from overlays, the extends property returns None.

```
>>> overlays = config.pop('shared.conf')
>>> print(config.extends)
None
```

CHAPTER

ELEVEN

ATTRIBUTE ACCESS TO CONFIG DATA

Config provides attribute-based access to its members. So long as the section, category, and key names conform to Python identifier naming rules, they can be accessed as attributes. The Python code will not compile, or will cause a runtime error if the object being accessed has a bad name.

Sections appear to be attributes of the config.

```
>>> config = schema.load(local_conf)
>>> config.section_1 is config['section_1']
True
```

Accessing an unknown section, or a section whose name is not a valid Python identifier will raise an AttributeError.

```
>>> config.section-5
Traceback (most recent call last):
  ...
AttributeError: No section or category named section.
```

Categories may be accessed as attributes too. The ICategory interface provides access to its sections as members.

```
>>> from lazr.config.interfaces import ICategory
>>> config_category = config.section_3
>>> verifyObject(ICategory, config_category)
True
>>> config_category.app_b is config['section_3.app_b']
True
```

Like a config, a category will raise an AttributeError if it does not have a section that matches the identifier name.

```
>>> config_category.no_such_section
Traceback (most recent call last):
  ...
AttributeError: No section named no_such_section.
```

Section keys can be accessed directly as members.

```
>>> print(config.section_1.key2)
sharing is fun
>>> print(config.section_3.app_b.key2)
changed
```

Accessing a non-existent section key as an attribute will raise an AttributeError.

```
>>> config.section_1.non_key
Traceback (most recent call last):
  ...
AttributeError: No section key named non_key.
```

CHAPTER

TWELVE

IMPLICIT DATA TYPING

The ImplicitTypeSchema can create configs that support implicit datatypes. The value of a Section key is automatically converted from `str` to the type the value appears to be. Implicit typing does not add any validation support; it adds type casting conveniences for the developer.

An ImplicitTypeSchema can be used to parse the same schema and conf files that Schema uses.

```
>>> from lazr.config import ImplicitTypeSchema
>>> implicit_schema = ImplicitTypeSchema(base_conf)
>>> verifyObject(IConfigSchema, implicit_schema)
True
```

The config loaded by ImplicitTypeSchema is the same class with the same sections as is made by Schema.

```
>>> implicit_config = implicit_schema.load(local_conf)
>>> implicit_config
<lazr.config...Config object at ...>
>>> config
<lazr.config...Config object at ...>

>>> sections = sorted(section.name for section in config)
>>> implicit_sections = sorted(
...     section.name for section in implicit_config)
>>> implicit_sections == sections
True

>>> verifyObject(ISection, implicit_config['section_3.app_b'])
True
```

But the type of sections in the config support implicit typing.

```
>>> implicit_config['section_3.app_b']
<lazr.config...ImplicitTypeSection object at ...>
```

ImplicitTypeSection, in contrast to Section, converts values that appear to be integer or boolean into ints and bools.

```
>>> config['section_3.app_b']['key1']
'17'
>>> implicit_config['section_3.app_b']['key1']
17

>>> config['section-2.app-b']['key1']
'True'
```

(continues on next page)

(continued from previous page)

```
>>> implicit_config['section-2.app-b']['key1']
True
```

The value is also converted when it is accessed as an attribute.

```
>>> implicit_config.section_3.app_b.key1
17
```

```
>>> implicit_config['section-2.app-b'].key1
True
```

ImplicitTypeSection uses a private method that employs heuristic rules to convert strings into simple types. It may return a str, bool, or int. When the argument is the word 'true' or 'false' (in any case), a bool is returned. Values like 'yes', 'no', '0', and '1' are not converted to bool.

```
>>> convert = implicit_config['section_1']._convert

>>> convert('false')
False
>>> convert('TRUE')
True
>>> convert('tRue')
True

>>> print(convert('yes'))
yes
>>> convert('1')
1
>>> print(convert('True or False'))
True or False
```

When the argument is the word `none`, `None` is returned. The token in the config means the key has no value.

```
>>> print(convert('none'))
None
>>> print(convert('None'))
None
>>> print(convert('nonE'))
None

>>> print(convert('none today'))
none today
>>> print(convert('nonevident'))
nonevident
```

When the argument is an unbroken sequence of numbers, an int is returned. The number may have a leading positive or negative. Octal and hex notation is not supported.

```
>>> convert('0')
0
>>> convert('2001')
2001
```

(continues on next page)

(continued from previous page)

```
>>> convert('-55')
-55
>>> convert('+404')
404
>>> convert('0100')
100

>>> print(convert('2001-01-01'))
2001-01-01
>>> print(convert('1000*60*5'))
1000*60*5
>>> print(convert('1000 * 60 * 5'))
1000 * 60 * 5
>>> print(convert('1,024'))
1,024
>>> print(convert('0.5'))
0.5
>>> print(convert('0x100'))
0x100
```

Multiline values are always strings, with white space (and line breaks) removed from the beginning and end.

```
>>> print(convert("""multiline value 1
...     multiline value 2"""))
multiline value 1
multiline value 2
```

CHAPTER

THIRTEEN

TYPE CONVERSION HELPERS

lazr.config provides a few helpers for doing explicit type conversion. These functions have to be imported and called explicitly on the configuration variable values.

13.1 Booleans

There is a helper for turning various strings into the boolean values `True` and `False`.

```
>>> from lazr.config import as_boolean
```

True values include (case-insensitively): true, yes, 1, on, enabled, and enable.

```
>>> for value in ('true', 'yes', 'on', 'enable', 'enabled', '1'):
...     print(value, '->', as_boolean(value))
...     print(value.upper(), '->', as_boolean(value.upper()))
true -> True
TRUE -> True
yes -> True
YES -> True
on -> True
ON -> True
enable -> True
ENABLE -> True
enabled -> True
ENABLED -> True
1 -> True
1 -> True
```

False values include (case-insensitively): false, no, 0, off, disabled, and disable.

```
>>> for value in ('false', 'no', 'off', 'disable', 'disabled', '0'):
...     print(value, '->', as_boolean(value))
...     print(value.upper(), '->', as_boolean(value.upper()))
false -> False
FALSE -> False
no -> False
NO -> False
off -> False
OFF -> False
disable -> False
```

(continues on next page)

(continued from previous page)

```
DISABLE -> False
disabled -> False
DISABLED -> False
0 -> False
0 -> False
```

Anything else is a error.

```
>>> as_boolean('cheese')
Traceback (most recent call last):
...
ValueError: Invalid boolean value: cheese
```

13.2 Host and port

There is a helper for converting from a `host:port` string to a 2-tuple of `(host, port)`.

```
>>> from lazr.config import as_host_port
>>> host, port = as_host_port('host:25')
>>> print(host, port)
host 25
```

The port string is optional, in which case, port 25 is the default (for historical reasons).

```
>>> host, port = as_host_port('host')
>>> print(host, port)
host 25
```

The default port can be overridden.

```
>>> host, port = as_host_port('host', default_port=22)
>>> print(host, port)
host 22
```

The default port is ignored if it is given in the value.

```
>>> host, port = as_host_port('host:80', default_port=22)
>>> print(host, port)
host 80
```

The host name is also optional, as denoted by a leading colon. When omitted, localhost is used.

```
>>> host, port = as_host_port(':80')
>>> print(host, port)
localhost 80
```

The default host name can be overridden though.

```
>>> host, port = as_host_port(':80', default_host='myhost')
>>> print(host, port)
myhost 80
```

The default host name is ignored if the value string contains it.

```
>>> host, port = as_host_port('yourhost:80', default_host='myhost')
>>> print(host, port)
yourhost 80
```

A ValueError occurs if the port number in the configuration value string is not an integer.

```
>>> as_host_port(':foo')
Traceback (most recent call last):
...
ValueError: invalid literal for int...foo...
```

13.3 User and group

A helper is provided for turning a `chown(1)`-style `user:group` specification into a 2-tuple of the user name and group name.

```
>>> from lazr.config import as_username_groupname
```

The value string must contain both a user name and group name, separated by a colon, otherwise an exception is raised.

```
>>> as_username_groupname('foo')
Traceback (most recent call last):
...
ValueError: ...
```

When both are given, the strings are returned unchanged or validated.

```
>>> user, group = as_username_groupname('person:group')
>>> print(user, group)
person group
```

Numeric values can be given, but they are not converted into their symbolic names.

```
>>> uid, gid = as_username_groupname('25:26')
>>> print(uid, gid)
25 26
```

By default the current user and group names are returned.

```
>>> import grp, os, pwd
>>> user, group = as_username_groupname()
>>> user == pwd.getpwuid(os.getuid()).pw_name
True
>>> group == grp.getgrgid(os.getgid()).gr_name
True
```

13.4 Time intervals

This converter accepts a range of *time interval specifications*, and returns a Python timedelta.

```
>>> from lazr.config import as_timedelta
```

The function converts from an integer to the equivalent number of seconds.

```
>>> as_timedelta('45s')
datetime.timedelta(...)
>>> print(as_timedelta('45s'))
0:00:45
```

The function also accepts suffixes `m` for minutes...

```
>>> print(as_timedelta('3m'))
0:03:00
```

...``h`` for hours...

```
>>> print(as_timedelta('2h'))
2:00:00
```

...and `d` for days...

```
>>> print(as_timedelta('4d'))
4 days, 0:00:00
```

...and `w` for weeks.

```
>>> print(as_timedelta('4w'))
28 days, 0:00:00
```

The function accepts a fractional number of seconds, indicating microseconds.

```
>>> print(as_timedelta('3.2s'))
0:00:03.200000
```

It also accepts any combination thereof.

```
>>> print(as_timedelta('3m22.5s'))
0:03:22.500000
>>> print(as_timedelta('4w2d9h3s'))
30 days, 9:00:03
```

But doesn't accept "weird" or duplicate combinations.

```
>>> as_timedelta('3s2s')
Traceback (most recent call last):
...
ValueError
>>> as_timedelta('2.9s4w')
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError
>>> as_timedelta('m')
Traceback (most recent call last):
...
ValueError
>>> as_timedelta('3m2')
Traceback (most recent call last):
...
ValueError
>>> as_timedelta('45')
Traceback (most recent call last):
...
ValueError
>>> as_timedelta('45wm')
Traceback (most recent call last):
...
ValueError
>>> as_timedelta('45z')
Traceback (most recent call last):
...
ValueError
```

13.5 Log levels

It's convenient to be able to use symbolic log level names when using `lazr.config` to configure the Python logger.

```
>>> from lazr.config import as_log_level
```

Any symbolic log level value is valid to use, case insensitively.

```
>>> for value in ('critical', 'error', 'warning', 'info',
...               'debug', 'notset'):
...     print(value, '->', as_log_level(value))
...     print(value.upper(), '->', as_log_level(value.upper()))
critical -> 50
CRITICAL -> 50
error -> 40
ERROR -> 40
warning -> 30
WARNING -> 30
info -> 20
INFO -> 20
debug -> 10
DEBUG -> 10
notset -> 0
NOTSET -> 0
```

Non-log levels cannot be used here.

```
>>> as_log_level('cheese')
Traceback (most recent call last):
...
AttributeError: ...
```

CHAPTER

FOURTEEN

OTHER DOCUMENTS

14.1 Hacking on lazr.config

These are guidelines for hacking on the lazr.config project. But first, please see the common hacking guidelines at:

http://dev.launchpad.net/Hacking

14.1.1 Getting help

If you find bugs in this package, you can report them here:

https://launchpad.net/lazr.config

If you want to discuss this package, join the team and mailing list here:

https://launchpad.net/~lazr-developers

or send a message to:

lazr-developers@lists.launchpad.net

14.2 Contributing

To run this project's tests, use tox.

To update the project's documentation you need to trigger a manual build on the project's dashboard on https://readthedocs.org.

14.3 NEWS for lazr.config

14.3.1 3.0 (2023-04-08)

- Add basic pre-commit configuration.
- Publish Documentation on Read the Docs.
- Apply inclusive naming via the woke pre-commit hook.
- Test using `zope.testrunner` rather than `nose`.
- Officially add support for Python 3.9, 3.10 and 3.11.
- Drop support for Python 2.6, 2.7, and 3.4.

- Apply `black` and `isort`.

14.3.2 2.2.3 (2021-01-26)

- Fix tests with zope.interface >= 5.0.0.
- Fix deprecation warning on Python >= 3.2. (LP: #1870199)

14.3.3 2.2.2 (2019-11-04)

- Officially add support for Python 3.7 and 3.8. The test suite required some changes since the *repr* of *datetime.timedelta* objects changed in 3.7.

14.3.4 2.2.1 (2017-10-20)

- Adjust versioning strategy to avoid importing pkg_resources, which is slow in large environments.

14.3.5 2.2 (2017-02-07)

- Fix tox import failure related to https://github.com/tox-dev/tox/issues/453 (LP: #1662701)
- Don't catch ImportErrors that might occur when importing lazr.config._config from lazr/config/__init__.py. It's unnecessary and masks legitimate ImportErrors of e.g. lazr.delegates.
- setup.py: nose is not an install_requires, so move this dependency to tox.ini. (LP: #1649726)
- tox.ini: Add the py36 environment and drop py32, py33. Ignore missing interpreters. Change to a temporary directory when running tox (to avoid the above tox bug). Invoke nose via -m instead of the mostly deprecated `python setup.py` approach.

14.3.6 2.1 (2015-01-05)

- Always use old-style namespace package registration in `lazr/__init__.py` since the mere presence of this file subverts PEP 420 style namespace packages. (LP: #1407816)
- For behavioral compatibility between Python 2 and 3, *strict=False* must be passed to the underlying *RawConfigParser* under Python 3. (LP: #1397779)

14.3.7 2.0.1 (2014-08-22)

- Drop the use of *distribute* in favor of *setuptools*. (LP: #1359926)
- Run the test suite with *tox*.

14.3.8 2.0 (2013-01-10)

- Ported to Python 3.
- Now more strict in its requirement of ASCII in config files.
- Category names are now sorted by default.

14.3.9 1.1.3 (2009-08-25)

- Fixed a build problem.

14.3.10 1.1.2 (2009-08-25)

- Got rid of a sys.path hack.

14.3.11 1.1.1 (2009-03-24)

- License clarification: only v3 of the LGPL is offered at this time, not subsequent versions.
- Build is updated to support Sphinx docs and other small changes.

14.3.12 1.1 (2009-01-05)

- Support for adding arbitrary sections in a configuration file, based on a .master section in the schema. The .master section allows admins to define configurations for an arbitrary number of processes. If the schema defines .master sections, then the conf file can contain sections that extend the .master section. These are like categories with templates except that the section names extending .master need not be named in the schema file. [Bug 310619]
- ConfigSchema now provides an interface for constructing the schema from a string. [Bug 309859]
- Added as_boolean() and as_log_level() type converters. [Bug 310782]
- getByCategory() accepts a default argument. If the category is missing, the default argument is returned. If the category is missing and no default argument is given, a NoCategoryError is raised, as before. [Bug 309988]

14.3.13 1.0 (2008-12-19)

- Initial release