ZConfig Documentation

Release 3.6.1.dev0

Zope Foundation

May 18, 2024

CONTENTS:

ZConfig is a Python library for creating extensible configuration documents (files). The configuration documents are written in a syntax reminiscent of that used by the Apache HTTP Server, while the configuration mechanism is itself configured using a schema specification written in XML.

ZConfig is used by projects such as the Zope application server and ZODB, and is easily used by other projects. ZConfig only relies on the Python standard library.

For information on reading and writing configuration documents, see *Reading and Writing Configurations*. For the extremely common usage of configuring the Python `logging` framework, see *Configuring Logging*.

For information on using ZConfig to create custom configurations for you projects, see *Developing With ZConfig*.

Development of ZConfig is hosted on GitHub.

CHAPTER

ONE

READING AND WRITING CONFIGURATIONS

This document describes how to read and write configurations in the ZConfig format.

1.1 Reading Configurations

For information on using ZConfig configuration documents in Python, see *ZConfig* and especially the example at *Basic Usage*.

For information about configuring the `logging` framework, see *Configuring Logging*.

1.2 Writing Configurations

Like the `ConfigParser` format, this format supports key-value pairs arranged in sections. Unlike the `ConfigParser` format, sections are typed and can be organized hierarchically. Additional files may be included if needed. Schema components not specified in the application schema can be imported from the configuration file. Though both formats are substantially line-oriented, this format is more flexible.

The intent of supporting nested section is to allow setting up the configurations for loosely-associated components in a container. For example, each process running on a host might get its configuration section from that host's section of a shared configuration file.

The top level of a configuration file consists of a series of inclusions, key-value pairs, and sections.

Comments can be added on lines by themselves. A comment has a # as the first non-space character and extends to the end of the line:

```
# This is a comment
```

An inclusion is expressed like this:

```
%include defaults.conf
```

The resource to be included can be specified by a relative or absolute URL, resolved relative to the URL of the resource the `%include` directive is located in.

A key-value pair is expressed like this:

```
key value
```

The key may include any non-white characters except for parentheses. The value contains all the characters between the key and the end of the line, with surrounding whitespace removed.

Since comments must be on lines by themselves, the # character can be part of a value:

```
key value # still part of the value
```

Sections may be either empty or non-empty. An empty section may be used to provide an alias for another section.

A non-empty section starts with a header, contains configuration data on subsequent lines, and ends with a terminator.

The header for a non-empty section has this form (square brackets denote optional parts):

<*section-type* **[name]**>

section-type and *name* all have the same syntactic constraints as key names.

The terminator looks like this:

</*section-type*>

The configuration data in a non-empty section consists of a sequence of one or more key-value pairs and sections. For example:

```
<my-section>
  key-1 value-1
  key-2 value-2

  <another-section>
      key-3 value-3
  </another-section>
</my-section>
```

(The indentation is used here for clarity, but is not required for syntactic correctness.)

The header for empty sections is similar to that of non-empty sections, but there is no terminator:

<*section-type* **[name]** />

1.2.1 Extending the Configuration Schema

As we'll see in *Writing Configuration Schema* what can be written in a configuration is controlled by schemas which can be built from **components**. These components can also be used to extend the set of implementations of objects the application can handle. What this means when writing a configuration is that third-party implementations of application object types can be used wherever those application types are used in the configuration, if there's a *ZConfig* component available for that implementation.

The configuration file can use an `%import` directive to load a named component:

```
%import Products.Ape
```

The text to the right of the `%import` keyword must be the name of a Python package; the *ZConfig* component provided by that package will be loaded and incorporated into the schema being used to load the configuration file. After the import, section types defined in the component may be used in the configuration.

More detail is needed for this to really make sense.

A schema may define section types which are **abstract**; these cannot be used directly in a configuration, but multiple concrete section types can be defined which **implement** the abstract types. Wherever the application allows an abstract type to be used, any concrete type which implements that abstract type can be used in an actual configuration.

The %import directive allows loading schema components which provide alternate concrete section types which implement the abstract types defined by the application. This allows third-party implementations of abstract types to be used in place of or in addition to implementations provided with the application.

Consider an example application which supports logging in the same way Zope 2 does. There are some parameters which configure the general behavior of the logging mechanism, and an arbitrary number of **log handlers** may be specified to control how the log messages are handled. Several log handlers are provided by the application. Here is an example logging configuration:

```
<eventlog>
  level verbose

  <logfile>
    path /var/log/myapp/events.log
  </logfile>
</eventlog>
```

A third-party component may provide a log handler to send high-priority alerts the system administrator's text pager or SMS-capable phone. All that's needed is to install the implementation so it can be imported by Python, and modify the configuration:

```
%import my.pager.loghandler

<eventlog>
  level verbose

  <logfile>
    path /var/log/myapp/events.log
  </logfile>

  <pager>
    number   1-800-555-1234
    message  Something broke!
  </pager>
</eventlog>
```

1.2.2 Other Examples

Other examples of configuration files can be found at *Using The Logging Components*.

1.2.3 Textual Substitution in Values

ZConfig provides a limited way to re-use portions of a value using simple string substitution. To use this facility, define named bits of replacement text using the %define directive, and reference these texts from values.

The syntax for %define is:

%define *name* [*value*]

The value of *name* must be a sequence of letters, digits, and underscores, and may not start with a digit; the namespace for these names is separate from the other namespaces used with *ZConfig*, and is case-insensitive. If *value* is omitted, it will be the empty string. If given, there must be whitespace between *name* and *value*; *value* will not include any whitespace on either side, just like values from key-value pairs.

Names must be defined before they are used, and may not be re-defined with a different value. All resources being parsed as part of a configuration share a single namespace for defined names.

References to defined names from configuration values use the syntax described for the *ZConfig.substitution* module. Configuration values which include a $ as part of the actual value will need to use $$ to get a single $ in the result.

The values of defined names are processed in the same way as configuration values, and may contain references to named definitions.

For example, the value for `key` will evaluate to `value`:

```
%define name value
key $name
```

1.2.4 Substitution in Values from Environment Variables

Values in *ZConfig* can be substituted from environment variables. It utilizes Pythons `os.getenv` to fetch the values. Syntax is a $ followed by round brackets (parentheses). In this example the variable key gets a value assigned from the enviroment named ENVKEY:

```
key $(ENVKEY)
```

Further details and examples are described in the *ZConfig.substitution* module.

CHAPTER

TWO

CONFIGURING LOGGING

One common use of ZConfig is to configure the Python `logging` framework. ZConfig provides one simple convenience function to do this:

`ZConfig.`**`configureLoggers`**`(`*text*`)`

Configure one or more loggers from configuration text.

Suppose we have the following logging configuration in a file called `simple-root-config.conf`:

```
<logger>
    level INFO
    <logfile>
       path STDOUT
       format %(levelname)s %(name)s %(message)s
    </logfile>
</logger>
```

We can load this file and pass its contents to `configureLoggers`:

```
from ZConfig import configureLoggers
with open('simple-root-config.conf') as f:
    configureLoggers(f.read())
```

When this returns, the root logger is configured to output messages logged at INFO or above to the console, as we can see in the following example:

```
>>> from logging import getLogger
>>> getLogger().info('We see an info message')
INFO root We see an info message
>>> getLogger().debug('We do not see a debug message')
```

A more common configuration would see STDOUT replaced with a path to the file into which log entries would be written.

Although loading configuration from a file is common, we could of course also pass a string literal to *`configureLoggers()`*. Any type of Python string (bytes or unicode) is acceptable.

2.1 Configuration Format

The configuration text is in the *ZConfig format* and supports comments and substitutions.

It can contain multiple <logger> elements, each of which can have any number of *handler elements*.

<logger> (ZConfig.components.logger.logger.LoggerFactory)

level (ZConfig.components.logger.datatypes.logging_level) (default: info)

Verbosity setting for the logger. Values must be a name of a level, or an integer in the range [0..50]. The names of the levels, in order of increasing verbosity (names on the same line are equivalent):

```
critical, fatal
error
warn, warning
info
blather
debug
trace
all
```

The special name "notset", or the numeric value 0, indicates that the setting for the parent logger should be used.

It is strongly recommended that names be used rather than numeric values to ensure that configuration files can be deciphered more easily.

zconfig.logger.handler*

Handlers to install on this logger. Each handler describes how logging events should be presented.

propagate (boolean) (default: true)

Indicates whether events that reach this logger should be propogated toward the root of the logger hierarchy. If true (the default), events will be passed to the logger's parent after being handled. If false, events will be handled and the parent will not be informed. There is not a way to control propogation by the severity of the event.

name (dotted-name)

The dotted name of the logger. This give it a location in the logging hierarchy. Most applications provide a specific set of subsystem names for which logging is meaningful; consult the application documentation for the set of names that are actually interesting for the application.

2.1.1 Examples

Here's the configuration we looked at above. It configures the root (unnamed) logger with one handler (<logfile>), operating at the INFO level:

```
<logger>
    level INFO
    <logfile>
       path STDOUT
       format %(levelname)s %(name)s %(message)s
    </logfile>
</logger>
```

We can configure a different logger in the hierarchy to use the DEBUG level at the same time as we configure the root logger. We're not specifying a handler for it, but the default `propagate` value will let the lower level logger use the root logger's handler:

```
<logger>
    level INFO
    <logfile>
       path STDOUT
       format %(levelname)s %(name)s %(message)s
    </logfile>
</logger>
<logger>
    name my.package
    level DEBUG
</logger>
```

If we load that configuration from `root-and-child-config.conf`, we can expect this behaviour:

```
>>> with open('root-and-child-config.conf') as f:
...     configureLoggers(f.read())
>>> getLogger().info('Here is another info message')
INFO root Here is another info message
>>> getLogger().debug('This debug message is hidden')
>>> getLogger('my.package').debug('The debug message for my.package shows')
DEBUG my.package The debug message for my.package shows
```

2.2 Log Handlers

Many of Python's built-in log handlers can be configured with ZConfig.

2.2.1 Files

The <logfile> handler writes to files or standard output or standard error (when the `path` is `STDOUT` or `STDERR` respectively). It configures a `logging.FileHandler` or `logging.StreamHandler`. When the `when` or `max-size` attributes are set, the files on disk will be rotated either at `set intervals` or when files `reach the set size`, respectively.

<logfile> (ZConfig.components.logger.handlers.FileHandlerFactory)

Example:

```
<logfile>
    path STDOUT
    format %(name)s %(message)s
</logfile>
```

formatter (dotted-name)

Logging formatter class.

The default is `logging.Formatter`.

One alternative is 'zope.exceptions.log.Formatter', which enhances exception tracebacks with information from `__traceback_info__` and `__traceback_supplement__` variables from each stack frame.

dateformat (string) (default: %Y-%m-%dT%H:%M:%S)

Timestamp format used for the 'asctime' field.

This is used with Python's `time.strftime()` function, so must be compatible with that function on the host platform.

level (ZConfig.components.logger.datatypes.logging_level) (default: notset)

Output level for the log handler.

Python standard logging levels are supported by name (case-insensitive), as are the following additional names:

- `all` (level 1)
- `trace` (level 5)
- `blather` (level 15)

These additional level names are not defined using `logging.addLevelName()`, though an application may do so.

Numeric values `0` through `50` (inclusive) are permitted.

style (ZConfig.components.logger.formatter.log_format_style) (default: classic)

Replacement mechanism to use with the `format` string.

The value must be one of `classic` (the default), `format`, `template`, or `safe-template`.

arbitrary-fields (boolean) (default: false)

If true, allow arbitrary fields in the log record object to be referenced from the `format` value.

This does not cause references to fields not present in the log record to be accepted; it only means unknown fields will not cause configuration of the formatter to be denied.

To get both effects, set this to `true` and use a `style` of `safe-template`.

path (string)

Path of the log file to write.

Specifying `STDOUT` or `STDERR` will cause the appropriate standard stream to be used instead of a file. In these cases, rotation, encoding, and delayed opening are not available, and will be considered configuration errors.

old-files (integer) (default: 0)

Number of old log files which will be retained when rotation is configured.

This must be set to a positive integer if rotation is configured. An error is generated if rotation is requested without setting `old-files`.

max-size (byte-size) (default: 0)

Target maximum size for a logfile; once a logfile reaches the maximum size, it will be rotated.

when (string)

Specification for specific time at which rotation should occur.

Allowed values are described for `logging.handlers.TimedRotatingFileHandler`; this value is passed through to the underlying handler.

interval (integer) (default: 0)

Frequency of rotation at the time specified by `when`.

If not specified, but `when` is specified, this will default to `1`.

If `when` is not specified, it is an error to specify `interval`.

format (ZConfig.components.logger.formatter.escaped_string) (default: ——\n%(asctime)s %(levelname)s %(name)s %(message)s)

Format string for log entries. This value is used to create an instance of the class identified by `formatter`.

The following escape characters are supported with the same replacements as in Python string literals:

```
\b  \f  \n  \r  \t
```

Field placeholders are checked to refer to the fields available in the `logging.LogRecord` instances created without extra fields. Referring to other fields will generate an error in loading the configuration, unless `arbitrary-fields` is true.

encoding (string)

Encoding for the underlying file.

If not specified, Python's default encoding handling is used.

This cannot be specified for `STDOUT` or `STDERR` destinations, and must be omitted in such cases.

delay (boolean) (default: false)

If true, opening of the log file will be delayed until a message is emitted. This avoids creating logfiles that may only be written to rarely or under special conditions.

This cannot be specified for `STDOUT` or `STDERR` destinations, and must be omitted in such cases.

2.2.2 The System Log

The <syslog> handler configures the `logging.handlers.SysLogHandler`.

<syslog> (ZConfig.components.logger.handlers.SyslogHandlerFactory)

formatter (dotted-name)

Logging formatter class.

The default is `logging.Formatter`.

One alternative is 'zope.exceptions.log.Formatter', which enhances exception tracebacks with information from `__traceback_info__` and `__traceback_supplement__` variables from each stack frame.

dateformat (string) (default: %Y-%m-%dT%H:%M:%S)

Timestamp format used for the 'asctime' field.

This is used with Python's `time.strftime()` function, so must be compatible with that function on the host platform.

level (ZConfig.components.logger.datatypes.logging_level) (default: notset)

Output level for the log handler.

Python standard logging levels are supported by name (case-insensitive), as are the following additional names:

- `all` (level 1)

- `trace` (level 5)
- `blather` (level 15)

These additional level names are not defined using `logging.addLevelName()`, though an application may do so.

Numeric values `0` through `50` (inclusive) are permitted.

style (ZConfig.components.logger.formatter.log_format_style) (default: classic)

Replacement mechanism to use with the `format` string.

The value must be one of `classic` (the default), `format`, `template`, or `safe-template`.

arbitrary-fields (boolean) (default: false)

If true, allow arbitrary fields in the log record object to be referenced from the `format` value.

This does not cause references to fields not present in the log record to be accepted; it only means unknown fields will not cause configuration of the formatter to be denied.

To get both effects, set this to `true` and use a `style` of `safe-template`.

facility (ZConfig.components.logger.handlers.syslog_facility) (default: user)

address (socket-address) (default: localhost:514)

format (ZConfig.components.logger.formatter.escaped_string) (default: %(name)s %(message)s)

2.2.3 Windows Event Log

On Windows, the <`win32-eventlog`> configures the `logging.handlers.NTEventLogHandler`.

<win32-eventlog> (ZConfig.components.logger.handlers.Win32EventLogFactory)

formatter (dotted-name)

Logging formatter class.

The default is `logging.Formatter`.

One alternative is 'zope.exceptions.log.Formatter', which enhances exception tracebacks with information from `__traceback_info__` and `__traceback_supplement__` variables from each stack frame.

dateformat (string) (default: %Y-%m-%dT%H:%M:%S)

Timestamp format used for the 'asctime' field.

This is used with Python's `time.strftime()` function, so must be compatible with that function on the host platform.

level (ZConfig.components.logger.datatypes.logging_level) (default: notset)

Output level for the log handler.

Python standard logging levels are supported by name (case-insensitive), as are the following additional names:

- `all` (level 1)
- `trace` (level 5)
- `blather` (level 15)

These additional level names are not defined using `logging.addLevelName()`, though an application may do so.

Numeric values `0` through `50` (inclusive) are permitted.

style (ZConfig.components.logger.formatter.log_format_style) (default: classic)

Replacement mechanism to use with the `format` string.

The value must be one of `classic` (the default), `format`, `template`, or `safe-template`.

arbitrary-fields (boolean) (default: false)

If true, allow arbitrary fields in the log record object to be referenced from the `format` value.

This does not cause references to fields not present in the log record to be accepted; it only means unknown fields will not cause configuration of the formatter to be denied.

To get both effects, set this to `true` and use a `style` of `safe-template`.

appname (string) (default: Zope)

format (ZConfig.components.logger.formatter.escaped_string) (default: %(levelname)s %(name)s %(message)s)

2.2.4 HTTP

The <<http-logger> element configures `logging.handlers.HTTPHandler`.

<http-logger> (ZConfig.components.logger.handlers.HTTPHandlerFactory)

formatter (dotted-name)

Logging formatter class.

The default is `logging.Formatter`.

One alternative is 'zope.exceptions.log.Formatter', which enhances exception tracebacks with information from `__traceback_info__` and `__traceback_supplement__` variables from each stack frame.

dateformat (string) (default: %Y-%m-%dT%H:%M:%S)

Timestamp format used for the 'asctime' field.

This is used with Python's `time.strftime()` function, so must be compatible with that function on the host platform.

level (ZConfig.components.logger.datatypes.logging_level) (default: notset)

Output level for the log handler.

Python standard logging levels are supported by name (case-insensitive), as are the following additional names:

- `all` (level 1)
- `trace` (level 5)
- `blather` (level 15)

These additional level names are not defined using `logging.addLevelName()`, though an application may do so.

Numeric values `0` through `50` (inclusive) are permitted.

style (ZConfig.components.logger.formatter.log_format_style) (default: classic)

Replacement mechanism to use with the `format` string.

The value must be one of `classic` (the default), `format`, `template`, or `safe-template`.

arbitrary-fields (boolean) (default: false)

If true, allow arbitrary fields in the log record object to be referenced from the `format` value.

This does not cause references to fields not present in the log record to be accepted; it only means unknown fields will not cause configuration of the formatter to be denied.

To get both effects, set this to `true` and use a `style` of `safe-template`.

url (ZConfig.components.logger.handlers.http_handler_url) (default: http://localhost/)

method (ZConfig.components.logger.handlers.get_or_post) (default: GET)

format (ZConfig.components.logger.formatter.escaped_string) (default: %(asctime)s %(levelname)s %(name)s %(message)s)

2.2.5 Email

ZConfig has support for Python's `logging.handlers.SMTPHandler` via the <email-notifier> handler.

<email-notifier> (ZConfig.components.logger.handlers.SMTPHandlerFactory)

Example:

```
<email-notifier>
  to sysadmin@example.com
  to john@example.com
  from zlog-user@example.com
  level fatal
  smtp-username john
  smtp-password johnpw
</email-notifier>
```

formatter (dotted-name)

Logging formatter class.

The default is `logging.Formatter`.

One alternative is 'zope.exceptions.log.Formatter', which enhances exception tracebacks with information from `__traceback_info__` and `__traceback_supplement__` variables from each stack frame.

dateformat (string) (default: %Y-%m-%dT%H:%M:%S)

Timestamp format used for the 'asctime' field.

This is used with Python's `time.strftime()` function, so must be compatible with that function on the host platform.

level (ZConfig.components.logger.datatypes.logging_level) (default: notset)

Output level for the log handler.

Python standard logging levels are supported by name (case-insensitive), as are the following additional names:

- `all` (level 1)

- `trace` (level 5)
- `blather` (level 15)

These additional level names are not defined using `logging.addLevelName()`, though an application may do so.

Numeric values `0` through `50` (inclusive) are permitted.

style (ZConfig.components.logger.formatter.log_format_style) (default: classic)

Replacement mechanism to use with the `format` string.

The value must be one of `classic` (the default), `format`, `template`, or `safe-template`.

arbitrary-fields (boolean) (default: false)

If true, allow arbitrary fields in the log record object to be referenced from the `format` value.

This does not cause references to fields not present in the log record to be accepted; it only means unknown fields will not cause configuration of the formatter to be denied.

To get both effects, set this to `true` and use a `style` of `safe-template`.

from (string)

to (*) (string)

subject (string) (default: Message from Zope)

smtp-server (inet-address) (default: localhost)

smtp-username (string)

User name to use for SMTP authentication. This can only be specified if `smtp-password` is also specified.

smtp-password (string)

Password to use for SMTP authentication. This can only be specified if `smtp-username` is also specified.

format (ZConfig.components.logger.formatter.escaped_string) (default: %(asctime)s %(levelname)s %(name)s %(message)s)

Format string for the email content.

The following escape characters are supported with the same replacements as in Python string literals:

```
\b  \f  \n  \r  \t
```

%-replacements are checked to refer to the fields available in the `logging.LogRecord` instances created without extra fields. Referring to other fields will generate an error in loading the configuration.

Emails will be sent without a `Content-Type` header, so the content will be interpreted as `text/plain`.

CHAPTER

THREE

DEVELOPING WITH ZCONFIG

3.1 Writing Configuration Schema

Configurations which use *ZConfig* are described using *"schema"*. A schema is a specification for the allowed structure and content of the configuration. *ZConfig* schema are written using a small XML-based language. The schema language allows the schema author to specify the names of the keys allowed at the top level and within sections, to define the types of sections which may be used (and where), the types of each values, whether a key or section must be specified or is optional, default values for keys, and whether a value can be given only once or repeatedly.

3.1.1 Writing Configuration Schema

Data types are searched in a special namespace defined by the data type registry. The default registry has slightly magical semantics: If the value can be matched to a standard data type when interpreted as a **basic-key**, the standard data type will be used. If that fails, the value must be a **dotted-name** containing at least one dot, and a conversion function will be sought using the *search()* method of the data type registry used to load the schema.

Schema Elements

For each element, the content model is shown, followed by a description of how the element is used, and then a list of the available attributes. For each attribute, the type of the value is given as either the name of a *ZConfig* datatype or an XML attribute value type. Familiarity with XML's Document Type Definition language is helpful.

The following elements are used to describe a schema:

```
<schema>
  description?, metadefault?, example?, import*, (sectiontype |
  abstracttype)*, (section | key | multisection | multikey)*
</schema>
```

Document element for a *ZConfig* schema.

extends (space-separated-url-references)
: A list of URLs of base schemas from which this section type will inherit key, section, and section type declarations. If omitted, this schema is defined using only the keys, sections, and section types contained within the `schema` element.

datatype (basic-key or dotted-name)
: The data type converter which will be applied to the value of this section. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. If any base schemas are listed in the `extends` attribute, the default value for this attribute comes from the base schemas. If the base schemas all use the same `datatype` , then that data type will be the default value for the extending schema. If there are no base schemas,

the default value is **null** , which means that the *ZConfig* section object will be used unconverted. If the base schemas have different `datatype` definitions, you must explicitly define the `datatype` in the extending schema.

handler (**basic-key**)

keytype (basic-key or dotted-name)
The data type converter which will be applied to keys found in this section. This can be used to constrain key values in different ways; two data types which may be especially useful are the **identifier** and **ipaddr-or-hostname** types. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. If any base schemas are listed in the `extends` attribute, the default value for this attribute comes from the base schemas. If the base schemas all use the same `keytype` , then that key type will be the default value for the extending schema. If there are no base schemas, the default value is **basic-key** . If the base schemas have different `keytype` definitions, you must explicitly define the `keytype` in the extending schema.

prefix (dotted-name)
Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts with the `schema` element if it hasn't been overridden by an inner element with a `prefix` attribute.

```
<description>
   PCDATA
</description>
```

Descriptive text explaining the purpose the container of the `description` element. Most other elements can contain a `description` element as their first child. At most one `description` element may appear in a given context.

format (NMTOKEN)

Optional attribute that can be added to indicate what conventions
are used to mark up the contained text. This is intended to serve as a hint for documentation extraction tools. Suggested values are:

Value	Content Format
`plain`	`text/plain`; blank lines separate paragraphs
`rest`	reStructuredText
`stx`	Classic Structured Text

```
<example>
   PCDATA
</example>
```

An example value. This serves only as documentation.

```
<metadefault>
  PCDATA
</metadefault>
```

A description of the default value, for human readers. This may include information about how a computed value is determined when the schema does not specify a default value.

```
<abstracttype>
   description?
</abstracttype>
```

Define an abstract section type.

name (basic-key)
The name of the abstract section type; required.

```
<sectiontype>
   description?, example?, (section | key | multisection | multikey)*
</sectiontype>
```

Define a concrete section type.

datatype (basic-key or dotted-name)
The data type converter which will be applied to the value of this section. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. If `datatype` is omitted and `extends` is used, the `datatype` from the section type identified by the `extends` attribute is used.

extends (basic-key)
The name of a concrete section type from which this section type acquires all key and section declarations. This type does **not** automatically implement any abstract section type implemented by the named section type. If omitted, this section is defined with only the keys and sections contained within the `sectiontype` element. The new section type is called a **derived** section type, and the type named by this attribute is called the **base** type. Values for the `datatype` and `keytype` attributes are acquired from the base type if not specified.

implements (basic-key)
The name of an abstract section type which this concrete section type implements. If omitted, this section type does not implement any abstract type, and can only be used if it is specified directly in a schema or other section type.

keytype (basic-key)
The data type converter which will be applied to keys found in this section. This can be used to constrain key values in different ways; two data types which may be especially useful are the **identifier** and **ipaddr-or-hostname** types. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set. The default value is **basic-key** . If `keytype` is omitted and `extends` is used, the `keytype` from the section type identified by the `extends` attribute is used.

name (basic-key)
The name of the section type; required.

prefix (dotted-name)
Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts in the `sectiontype` element. If omitted, the prefix specified by a containing context is used if specified.

```
<import>
  EMPTY
</import>
```

Import a schema component. Exactly one of the attributes `package` and `src` must be specified.

file (file name without directory information}
Name of the component file within a package; if not specified, 'component.xml' is used. This may only be given when `package` is used. (The 'component.xml' file is always used when importing via `%import` from a configuration file.)

package (dotted-suffix)
Name of a Python package that contains the schema component being imported. The component will be loaded from the file identified by the `file` attribute, or 'component.xml' if `file` is not specified. If the package name given starts with a dot (`.`), the name used will be the current prefix and the value of this attribute concatenated.

src (url-reference)
URL to a separate schema which can provide useful types. The referenced resource must contain a schema,

not a schema component. Section types defined or imported by the referenced schema are added to the schema containing the `import` ; top-level keys and sections are ignored.

```
<key>
  description?, example?, metadefault?, default*
</key>
```

A `key` element is used to describe a key-value pair which may occur at most once in the section type or top-level schema in which it is listed.

attribute (identifier)
The name of the Python attribute which this key should be the value of on a `SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the key name to underscores.

datatype (basic-key or dotted-name)
The data type converter which will be applied to the value of this key. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set.

default (string)
If the key-value pair is optional and this attribute is specified, the value of this attribute will be converted using the appropriate data type converter and returned to the application as the configured value. This attribute may not be specified if the `required` attribute is `yes`.

handler (**basic-key**)

name (basic-key)
The name of the key, as it must be given in a configuration instance, or `*`. If the value is `*`, any name not already specified as a key may be used, and the configuration value for the key will be a dictionary mapping from the key name to the value. In this case, the `attribute` attribute must be specified, and the data type for the key will be applied to each key which is found.

required (`yes|no`)
Specifies whether the configuration instance is required to provide the key. If the value is `yes`, the `default` attribute may not be specified and an error will be reported if the configuration instance does not specify a value for the key. If the value is `no` (the default) and the configuration instance does not specify a value, the value reported to the application will be that specified by the `default` attribute, if given, or `None`.

```
<multikey>
  description?, example?, metadefault?, default*
</multikey>
```

A `multikey` element is used to describe a key-value pair which may occur any number of times in the section type or top-level schema in which it is listed.

attribute (identifier)
The name of the Python attribute which this key should be the value of on a :class`SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the key name to underscores.

datatype (basic-key or dotted-name)
The data type converter which will be applied to the value of this key. If the value is a **dotted-name** that begins with a period, the value of `prefix` will be pre-pended, if set.

handler (**basic-key**)

name (basic-key)
The name of the key, as it must be given in a configuration instance, or `+`. If the value is `+`, any name not already specified as a key may be used, and the configuration value for the key will be a dictionary mapping from the key

name to the value. In this case, the `attribute` attribute must be specified, and the data type for the key will be applied to each key which is found.

required (`yes`|`no`)
Specifies whether the configuration instance is required to provide the key. If the value is `yes`, no `default` elements may be specified and an error will be reported if the configuration instance does not specify at least one value for the key. If the value is `no` (the default) and the configuration instance does not specify a value, the value reported to the application will be a list containing one element for each `default` element specified as a child of the `multikey` . Each value will be individually converted according to the `datatype` attribute.

```
<default>
  PCDATA
</default>
```

Each `default` element specifies a single default value for a `multikey` . This element can be repeated to produce a list of individual default values. The text contained in the element will be passed to the datatype conversion for the `multikey` .

key (key type of the containing sectiontype}
Key to associate with the default value. This is only used for defaults of a `key` or `multikey` with a `name` of +; in that case this attribute is required. It is an error to use the `key` attribute with a `default` element for a `multikey` with a name other than +.

Warning: The datatype of this attribute is that of the section type **containing** the actual keys, not necessarily that of the section type which defines the key. If a derived section overrides the key type of the base section type, the actual key type used is that of the derived section.

This can lead to confusing errors in schemas, though the *ZConfig* package checks for this when the schema is loaded. This situation is particularly likely when a derived section type uses a key type which collapses multiple default keys which were not collapsed by the base section type.

Consider this example schema:

```
<schema>
  <sectiontype name="base" keytype="identifier">
     <key name="+" attribute="mapping">
       <default key="foo">some value</default>
       <default key="FOO">some value</default>
     </key>
   </sectiontype>

   <sectiontype name="derived" keytype="basic-key"
         extends="base"/>

   <section type="derived" name="*" attribute="section"/>
 </schema>
```

When this schema is loaded, a set of defaults for the **derived** section type is computed. Since **basic-key** is case-insensitive (everything is converted to lower case), `foo` and `Foo` are both converted to `foo`, which clashes since `key` only allows one value for each key.

```
<section>
  description?, example?
</section>
```

A `section` element is used to describe a section which may occur at most once in the section type or top-level schema in which it is listed.

attribute (identifier)
The name of the Python attribute which this section should be the value of on a `SectionValue` instance. This must be unique within the immediate contents of a section type or schema. If this attribute is not specified, an attribute name will be computed by converting hyphens in the section name to underscores, in which case the `name` attribute may not be * or +.

handler (**basic-key**)

name (basic-key)
The name of the section, as it must be given in a configuration instance, *, or +. If the value is * or this attribute is omitted, any name not already specified as a key may be used. If the value is * or +, the `attribute` attribute must be specified. If the value is *, any name is allowed, or the name may be omitted. If the value is +, any name is allowed, but some name must be provided.

required (`yes`|`no`)
Specifies whether the configuration instance is required to provide the section. If the value is `yes`, an error will be reported if the configuration instance does not include the section. If the value is `no` (the default) and the configuration instance does not include the section, the value reported to the application will be `None`.

type (basic-key)
The section type which matching sections must implement. If the value names an abstract section type, matching sections in the configuration file must be of a type which specifies that it implements the named abstract type. If the name identifies a concrete type, the section type must match exactly.

```
<multisection>
  description?, example?
</multisection>
```

A `multisection` element is used to describe a section which may occur any number of times in the section type or top-level schema in which it is listed.

attribute (identifier)
The name of the Python attribute which matching sections should be the value of on a `SectionValue` instance. This is required and must be unique within the immediate contents of a section type or schema. The `SectionValue` instance will contain a list of matching sections.

handler (**basic-key**)

name (basic-key)
For a `multisection` , any name not already specified as a key may be used. If the value is * or +, the `attribute` attribute must be specified. If the value is * or this attribute is omitted, any name is allowed, or the name may be omitted. If the value is +, any name is allowed, but some name must be provided. No other value for the `name` attribute is allowed for a `multisection` .

required (`yes`|`no`)
Specifies whether the configuration instance is required to provide at least one matching section. If the value is `yes`, an error will be reported if the configuration instance does not include the section. If the value is `no` (the default) and the configuration instance does not include the section, the value reported to the application will be `None`.

type (basic-key)
The section type which matching sections must implement. If the value names an abstract section type, matching sections in the configuration file must be of types which specify that they implement the named abstract type. If the name identifies a concrete type, the section type must match exactly.

Schema Components

ZConfig supports schema components that can be provided by disparate components, and allows them to be knit together into concrete schema for applications. Components cannot add additional keys or sections in the application schema.

A schema *component* is allowed to define new abstract and section types. Components are identified using a dotted-name, similar to a Python module name. For example, one component may be `zodb.storage`.

Schema components are stored alongside application code since they directly reference datatype code. Schema components are provided by Python packages. The component definition is normally stored in the file 'component.xml'; an alternate filename may be specified using the `file` attribute of the `import` element. Components imported using the `%import` keyword from a configuration file must be named 'component.xml'. The component defines the types provided by that component; it must have a `component` element as the document element.

The following element is used as the document element for schema components. Note that schema components do not allow keys and sections to be added to the top-level of a schema; they serve only to provide type definitions.

```
<component>
  description?, (abstracttype | sectiontype)*
</component>
```

The top-level element for schema components.

prefix (dotted-name)
: Prefix to be pre-pended in front of partial dotted-names that start with a period. The value of this attribute is used in all contexts within the `component` element if it hasn't been overridden by an inner element with a `prefix` attribute.

Referring to Files in Packages

The `extends` attribute of the `schema` element is used to refer to files containing base schema; sometimes it makes sense to refer to a base schema relative to the Python package that provides it. For this purpose, ZConfig supports the special `package:` URL scheme.

The `package:` URL scheme is straightforward, and contains three parts: the scheme name, the package name, and a relative path. The relative path is searched for using the named package's `__path__` if it's a conventional filesystem package, or using the package's loader if that supports resource access (such as the loader for eggs and other ZIP-file based packages).

The basic form of the `package:` URL is:

`package:`*`package.name`*`:`*`relative-path`*

The package name must be fully specified; the current prefix, if any, is not used. If the named package is contained in an egg or ZIP file, the resource identified by the relative path must reside in the same egg or ZIP file.

The `package:` URL scheme is generally available everywhere ZConfig supports loading text from URLs directly, but applications using ZConfig do not automatically acquire general support for this.

3.1.2 Schema Document Type Definition

The following is the XML Document Type Definition for *ZConfig* schema:

```
<!--
  ****************************************************************************
  Copyright (c) 2002, 2003 Zope Foundation and Contributors.
  All Rights Reserved.

  This software is subject to the provisions of the Zope Public License,
  Version 2.1 (ZPL).  A copy of the ZPL should accompany this distribution.
  THIS SOFTWARE IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED
  WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
  WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS
  FOR A PARTICULAR PURPOSE.
  ****************************************************************************

  Please note that not all documents that conform to this DTD are
  legal ZConfig schema.  The ZConfig reference manual describes many
  constraints that are important to understanding ZConfig schema.
  -->

<!-- DTD for ZConfig schema documents. -->

<!ELEMENT schema (description?, metadefault?, example?,
                  import*,
                  (sectiontype | abstracttype)*,
                  (section | key | multisection | multikey)*)>
<!ATTLIST schema
          extends    NMTOKEN  #IMPLIED
          prefix     NMTOKEN  #IMPLIED
          handler    NMTOKEN  #IMPLIED
          keytype    NMTOKEN  #IMPLIED
          datatype   NMTOKEN  #IMPLIED>

<!ELEMENT component (description?, (sectiontype | abstracttype)*)>
<!ATTLIST component
          prefix     NMTOKEN  #IMPLIED>

<!ELEMENT import EMPTY>
<!ATTLIST import
          file       CDATA    #IMPLIED
          package    NMTOKEN  #IMPLIED
          src        CDATA    #IMPLIED>

<!ELEMENT description (#PCDATA)*>
<!ATTLIST description
          format     NMTOKEN  #IMPLIED>

<!ELEMENT metadefault (#PCDATA)*>
<!ELEMENT example     (#PCDATA)*>

<!ELEMENT sectiontype (description?, example?
```

(continues on next page)

(continued from previous page)

```
                        (section | key | multisection | multikey)*)>
<!ATTLIST sectiontype
          name        NMTOKEN  #REQUIRED
          prefix      NMTOKEN  #IMPLIED
          keytype     NMTOKEN  #IMPLIED
          datatype    NMTOKEN  #IMPLIED
          implements  NMTOKEN  #IMPLIED
          extends     NMTOKEN  #IMPLIED>

<!ELEMENT abstracttype (description?)>
<!ATTLIST abstracttype
          name        NMTOKEN  #REQUIRED
          prefix      NMTOKEN  #IMPLIED>

<!ELEMENT default    (#PCDATA)*>
<!ATTLIST default
          key         CDATA    #IMPLIED>

<!ELEMENT key (description?, metadefault?, example?, default*)>
<!ATTLIST key
          name        CDATA    #REQUIRED
          attribute   NMTOKEN  #IMPLIED
          datatype    NMTOKEN  #IMPLIED
          handler     NMTOKEN  #IMPLIED
          required    (yes|no) "no"
          default     CDATA    #IMPLIED>

<!ELEMENT multikey (description?, metadefault?, example?, default*)>
<!ATTLIST multikey
          name        CDATA    #REQUIRED
          attribute   NMTOKEN  #IMPLIED
          datatype    NMTOKEN  #IMPLIED
          handler     NMTOKEN  #IMPLIED
          required    (yes|no) "no">

<!ELEMENT section (description?, example?)>
<!ATTLIST section
          name        CDATA    "*"
          attribute   NMTOKEN  #IMPLIED
          type        NMTOKEN  #REQUIRED
          handler     NMTOKEN  #IMPLIED
          required    (yes|no) "no">

<!ELEMENT multisection (description?, example?)>
<!ATTLIST multisection
          name        CDATA    "*"
          attribute   NMTOKEN  #IMPLIED
          type        NMTOKEN  #REQUIRED
          handler     NMTOKEN  #IMPLIED
          required    (yes|no) "no">
```

3.2 Standard ZConfig Datatypes

There are a number of data types which can be identified using the `datatype` attribute on `key` , `sectiontype`, and `schema` elements. Applications may extend the set of datatypes by calling the *register()* method of the data type registry being used or by using Python dotted-names to refer to conversion routines defined in code.

The following data types are provided by the default type registry.

basic-key
The default data type for a key in a ZConfig configuration file. The result of conversion is always lower-case, and matches the regular expression `[a-z][-._a-z0-9]*`.

boolean
Convert a human-friendly string to a boolean value. The names `yes`, `on`, and `true` convert to `True`, while `no`, `off`, and `false` convert to `False`. Comparisons are case-insensitive. All other input strings are disallowed.

byte-size
A specification of a size, with byte multiplier suffixes (for example, `128MB`). Suffixes are case insensitive and may be `KB`, `MB`, or `GB`

dotted-name
A string consisting of one or more **identifier** values separated by periods (`.`).

dotted-suffix
A string consisting of one or more **identifier** values separated by periods (`.`), possibly prefixed by a period. This can be used to indicate a dotted name that may be specified relative to some base dotted name.

existing-dirpath
Validates that the directory portion of a pathname exists. For example, if the value provided is '/foo/bar', '/foo' must be an existing directory. No conversion is performed.

existing-directory
Validates that a directory by the given name exists on the local filesystem. No conversion is performed.

existing-file
Validates that a file by the given name exists. No conversion is performed.

existing-path
Validates that a path (file, directory, or symlink) by the given name exists on the local filesystem. No conversion is performed.

float
A Python float. `Inf`, `-Inf`, and `NaN` are not allowed.

identifier
Any valid Python identifier.

inet-address
An Internet address expressed as a `(hostname, port)` pair. If only the port is specified, the default host will be returned for *hostname*. The default host is `localhost` on Windows and the empty string on all other platforms. If the port is omitted, `None` will be returned for *port*. IPv6 addresses can be specified in colon-separated notation; if both host and port need to be specified, the bracketed form (`[addr]:port`) must be used.

inet-binding-address
An Internet address expressed as a `(hostname, port)` pair. The address is suitable for binding a socket. If only the port is specified, the default host will be returned for *hostname*. The default host is the empty string on all platforms. If the port is omitted, `None` will be returned for *port*.

inet-connection-address
An Internet address expressed as a `(hostname, port)` pair. The address is suitable for connecting a socket to

a server. If only the port is specified, `'127.0.0.1'` will be returned for *hostname*. If the port is omitted, `None` will be returned for *port*.

integer
Convert a value to an integer. This will be a Python `int` if the value is in the range allowed by :class`int`, otherwise a Python `long` is returned.

ipaddr-or-hostname
Validates a valid IP address or hostname. If the first character is a digit, the value is assumed to be an IP address. If the first character is not a digit, the value is assumed to be a hostname. Strings containing colons are considered IPv6 address. Hostnames are converted to lower case.

locale
Any valid locale specifier accepted by the available `locale.setlocale()` function. Be aware that only the `'C'` locale is supported on some platforms.

null
No conversion is performed; the value passed in is the value returned. This is the default data type for section values.

port-number
Returns a valid port number as an integer. Validity does not imply that any particular use may be made of the port, however. For example, port number lower than 1024 generally cannot be bound by non-root users.

socket-address
An address for a socket. The converted value is an object providing two attributes. `family` specifies the address family (`socket.AF_INET` or `socket.AF_UNIX`), with `None` instead of `AF_UNIX` on platforms that don't support it. The `address` attribute will be the address that should be passed to the socket's `bind()` method. If the family is `AF_UNIX`, the specific address will be a pathname; if the family is `AF_INET`, the second part will be the result of the **inet-address** conversion.

string
Returns the input value as a string. If the source is a Unicode string, this implies that it will be checked to be simple 7-bit ASCII. This is the default data type for values in configuration files.

time-interval
A specification of a time interval in seconds, with multiplier suffixes (for example, `12h`). Suffixes are case insensitive and may be `s` (seconds), `m` (minutes), `h` (hours), or `d` (days).

timedelta
Similar to the **time-interval**, this data type returns a Python `datetime.timedelta` object instead of a float. The set of suffixes recognized by **timedelta** are: `w` (weeks), `d` (days), `h` (hours), `m` (minutes), `s` (seconds). Values may be floats, for example: `4w 2.5d 7h 12m 0.001s`.

3.3 Standard `ZConfig` Schema Components

ZConfig provides a few convenient schema components as part of the package. These may be used directly or can serve as examples for creating new components.

3.3.1 ZConfig.components.basic

The `ZConfig.components.basic` package provides small components that can be helpful in composing application-specific components and schema. There is no large functionality represented by this package. The default component provided by this package simply imports all of the smaller components. This can be imported using:

```
<import package="ZConfig.components.basic"/>
```

Each of the smaller components is documented directly; importing these selectively can reduce the time it takes to load a schema slightly, and allows replacing the other basic components with alternate components (by using different imports that define the same type names) if desired.

The Mapping Section Type

There is a basic section type that behaves like a simple Python mapping; this can be imported directly using:

```
<import package="ZConfig.components.basic" file="mapping.xml"/>
```

This defines a single section type, **ZConfig.basic.mapping**. When this is used, the section value is a Python dictionary mapping keys to string values.

This type is intended to be used by extending it in simple ways. The simplest is to create a new section type name that makes more sense for the application:

```
<import package="ZConfig.components.basic" file="mapping.xml"/>

<sectiontype name="my-mapping"
             extends="ZConfig.basic.mapping"
             />

<section name="*"
         type="my-mapping"
         attribute="map"
         />
```

This allows a configuration to contain a mapping from **basic-key** names to string values like this:

```
<my-mapping>
  This that
  and the other
</my-mapping>
```

The value of the configuration object's `map` attribute would then be the dictionary:

```
{'this': 'that',
 'and': 'the other',
 }
```

(Recall that the **basic-key** data type converts everything to lower case.)

Perhaps a more interesting application of **ZConfig.basic.mapping** is using the derived type to override the `keytype` . If we have the conversion function:

```
def email_address(value):
    userid, hostname = value.split("@", 1)
    hostname = hostname.lower()  # normalize what we know we can
    return "%s@%s" % (userid, hostname)
```

then we can use this as the key type for a derived mapping type:

```
<import package="ZConfig.components.basic" file="mapping.xml"/>

<sectiontype name="email-users"
             extends="ZConfig.basic.mapping"
             keytype="mypkg.datatypes.email_address"
             />

<section name="*"
         type="email-users"
         attribute="email_users"
         />
```

3.4 Logging Components

The `ZConfig.components.logger` package provides configuration support for the `logging` package in Python's standard library. This component can be imported using:

```
<import package="ZConfig.components.logger"/>
```

This component defines two abstract types and several concrete section types. These can be imported as a unit, as above, or as four smaller components usable in creating alternate logging packages.

The first of the four smaller components contains the abstract types, and can be imported using:

```
<import package="ZConfig.components.logger" file="abstract.xml"/>
```

The two abstract types imported by this are:

ZConfig.logger.log
: Logger objects are represented by this abstract type.

ZConfig.logger.handler
: Each logger object can have one or more "handlers" associated with them. These handlers are responsible for writing logging events to some form of output stream using appropriate formatting. The output stream may be a file on a disk, a socket communicating with a server on another system, or a series of `syslog` messages. Section types which implement this type represent these handlers.

The second and third of the smaller components provides section types that act as factories for `logging.Logger` objects. These can be imported using:

```
<import package="ZConfig.components.logger" file="eventlog.xml"/>
<import package="ZConfig.components.logger" file="logger.xml"/>
```

The types defined in these components implement the **ZConfig.logger.log** abstract type. The 'eventlog.xml' component defines an **eventlog** type which represents the root logger from the the `logging` package (the return value of `logging.getLogger()`), while the 'logger.xml' component defines a **logger** section type which represents a named logger.

The third of the smaller components provides section types that are factories for `logging.Handler` objects. This can be imported using:

```
<import package="ZConfig.components.logger" file="handlers.xml"/>
```

The types defined in this component implement the **ZConfig.logger.handler** abstract type.

The configuration objects provided by both the logger and handler types are factories for the finished loggers and handlers. These factories should be called with no arguments to retrieve the logger or log handler objects. Calling the factories repeatedly will cause the same objects to be returned each time, so it's safe to simply call them to retrieve the objects.

The factories for the logger objects, whether the **eventlog** or **logger** section type is used, provide a `reopen()` method which may be called to close any log files and re-open them. This is useful when using a UNIX signal to effect log file rotation: the signal handler can call this method, and not have to worry about what handlers have been registered for the logger. There is also a function in the `ZConfig.components.logger.loghandler` module that re-opens all open log files created using ZConfig configuration:

`ZConfig.components.logger.loghandler.`**`reopenFiles`**`()`

> Closes and re-opens all the log files held open by handlers created by the factories for `logfile` sections. This is intended to help support log rotation for applications.

3.4.1 Using The Logging Components

Building an application that uses the logging components is fairly straightforward. The schema needs to import the relevant components and declare their use:

```
<schema>
  <import package="ZConfig.components.logger" file="eventlog.xml"/>
  <import package="ZConfig.components.logger" file="handlers.xml"/>

  <section type="eventlog" name="*" attribute="eventlog"
           required="yes"/>
</schema>
```

In the application, the schema and configuration file should be loaded normally. Once the configuration object is available, the logger factory should be called to configure Python's `logging` package:

```
import os
import ZConfig

def run(configfile):
  schemafile = os.path.join(os.path.dirname(__file__), "schema.xml")
  schema = ZConfig.loadSchema(schemafile)
  config, handlers = ZConfig.loadConfig(schema, configfile)

  # configure the logging package:
  config.eventlog()

  # now do interesting things
```

An example configuration file for this application may look like this:

```
<eventlog>
  level  info

  <logfile>
    path        /var/log/myapp
    format      %(asctime)s %(levelname)s %(name)s %(message)s
    # locale-specific date/time representation
    dateformat  %c
  </logfile>

  <syslog>
    level    error
    address  syslog.example.net:514
    format   %(levelname)s %(name)s %(message)s
  </syslog>
</eventlog>
```

Refer to the `logging.LogRecord` documentation for the names available in the message format strings (the `format` key in the log handlers). The date format strings (the `dateformat` key in the log handlers) are the same as those accepted by the `time.strftime()` function.

3.4.2 Configuring The Logging Components

For reference documentation on the available handlers, see *Log Handlers*.

3.5 Using Components to Extend Schema

It is possible to use schema components and the `%import` construct to extend the set of section types available for a specific configuration file, and allow the new components to be used in place of standard components.

The key to making this work is the use of abstract section types. Wherever the original schema accepts an abstract type, it is possible to load new implementations of the abstract type and use those instead of, or in addition to, the implementations loaded by the original schema.

Abstract types are generally used to represent interfaces. Sometimes these are interfaces for factory objects, and sometimes not, but there's an interface that the new component needs to implement. What interface is required should be documented in the `description` element in the `abstracttype` element; this may be by reference to an interface specified in a Python module or described in some other bit of documentation.

The following things need to be created to make the new component usable from the configuration file:

1. An implementation of the required interface.
2. A schema component that defines a section type that contains the information needed to construct the component.
3. A `datatype` function that converts configuration data to an instance of the component.

For simplicity, let's assume that the implementation is defined by a Python class.

The example component we build here will be in the `noise` package, but any package will do. Components loadable using `%import` must be contained in the `component.xml` file; alternate filenames may not be selected by the `%import` construct.

Create a ZConfig component that provides a section type to support your component. The new section type must declare that it implements the appropriate abstract type; it should probably look something like this:

```
<component prefix="noise.server">
 <import package="ZServer"/>

 <sectiontype name="noise-generator"
              implements="ZServer.server"
              datatype=".NoiseServerFactory">

   <!-- specific configuration data should be described here -->

   <key name="port"
        datatype="port-number"
        required="yes">
     <description>
       Port number to listen on.
     </description>
   </key>

   <key name="color"
        datatype=".noise_color"
        default="white">
     <description>
       Silly way to specify a noise generation algorithm.
     </description>
   </key>

 </sectiontype>
</component>
```

This example uses one of the standard ZConfig datatypes, **port-number**, and requires two additional types to be provided by the `noise.server` module: `NoiseServerFactory` and `noise_color()`.

The `noise_color()` function is a datatype conversion for a key, so it accepts a string and returns the value that should be used:

```
_noise_colors = {
    # color -> r,g,b
    'white': (255, 255, 255),
    'pink':  (255, 182, 193),
    }

def noise_color(string):
    if string in _noise_colors:
        return _noise_colors[string]
    else:
        raise ValueError('unknown noise color: %r' % string)
```

`NoiseServerFactory` is a little different, as it's the datatype function for a section rather than a key. The parameter isn't a string, but a section value object with two attributes, `port` and `color`.

Since the **ZServer.server** abstract type requires that the component returned is a factory object, the datatype function can be implemented at the constructor for the class of the factory object. (If the datatype function could select different implementation classes based on the configuration values, it makes more sense to use a simple function that returns the appropriate implementation.)

A class that implements this datatype might look like this:

```
from ZServer.datatypes import ServerFactory
from noise.generator import WhiteNoiseGenerator, PinkNoiseGenerator

class NoiseServerFactory(ServerFactory):

  def __init__(self, section):
      # host and ip will be initialized by ServerFactory.prepare()
      self.host = None
      self.ip = None
      self.port = section.port
      self.color = section.color

  def create(self):
      if self.color == 'white':
          generator = WhiteNoiseGenerator()
      else:
          generator = PinkNoiseGenerator()
      return NoiseServer(self.ip, self.port, generator)
```

You'll need to arrange for the package containing this component to be available on Python's `sys.path` before the configuration file is loaded; this is mostly easily done by manipulating the `PYTHONPATH` environment variable.

Your configuration file can now include the following to load and use your new component:

```
%import noise

<noise-generator>
  port 1234
  color white
</noise-generator>
```

3.6 Documenting Components

ZConfig includes a docutils directive for documenting components and schemas that you create. This directive can function as a Sphinx extension if you include `ZConfig.sphinx` in the `extensions` value of your Sphinx configuration:

```
extensions = [
  'sphinx.ext.autodoc',
  'sphinx.ext.interpsphinx',
  'ZConfig.sphinx',
]
```

There is one directive:

`.. zconfig::` <package-name>

New in version 3.2.0.

Document the components or schema found in the Python package *package-name*.

By default, the contents of `component.xml` will be documented. You can specify the `:file:` option to choose a different file from that package. This file can refer to a schema or component definition.

Each component will have its name, type, and default value documented. The description of the component will be rendered as reStructuredText (and so can use directives like `py:class:` and `py:meth:` to perform cross

references). Any example for the component will be rendered as a pre-formatted block.

All ZConfig components reachable will be documented, in the order in which they are found. Often times, if your component extends other components, this will produce too much documentation (it will document the components you are extending in addition to the unique aspects of your component). You can use the `:members:` and `:excluded-members:` options to limit this.

Both of these options take a space-separated list of component names. These options can be used together. When `:members:` is given, only items that are explicitly named, or that are reachable from such items, are documented. The `:excluded-members:` option overrides this, causing any such members to be explicitly excluded.

These options are also useful for breaking the description of a component up into multiple distinct sections, with narrative documentation between them. For example, to document the main logger component provided by ZConfig separately from each type of handler ZConfig provides, the document might look like this:

```
You can configure a logger and logging level with ZConfig:

.. zconfig:: ZConfig.components.logger
    :members: ZConfig.logger.base-logger
    :excluded-members: zconfig.logger.handler

ZConfig supports multiple different types of handlers for a given logger:

.. zconfig:: ZConfig.components.logger
    :members: zconfig.logger.handler
```

CHAPTER

FOUR

PYTHON API

4.1 ZConfig — Basic configuration support

4.1.1 Functions

The main *ZConfig* package exports these convenience functions:

ZConfig.**loadConfig**(*schema*, *url*, *overrides=()*)

Load and return a configuration from a URL or pathname given by *url*.

url may be a URL, absolute pathname, or relative pathname. Fragment identifiers are not supported. *schema* is a reference to a schema loaded by *loadSchema()* or *loadSchemaFile()*.

The return value is a tuple containing the configuration object and a composite handler that, when called with a name-to-handler mapping, calls all the handlers for the configuration.

The optional *overrides* argument represents information derived from command-line arguments. If given, it must be either a sequence of value specifiers, or `None`. A "value specifier" is a string of the form `optionpath=value`, for example, `some/path/to/key=value`.

See also:

ExtendedConfigLoader.addOption()
For information on the format of value specifiers.

ConfigLoader
For information about loading configs.

BaseLoader.loadURL()
For information about the format of *url*

ZConfig.**loadConfigFile**(*schema*, *file*, *url=None*, *overrides=()*)

Load and return a configuration from an opened file object.

If *url* is omitted, one will be computed based on the `name` attribute of *file*, if it exists. If no URL can be determined, all `%include` statements in the configuration must use absolute URLs. *schema* is a reference to a schema loaded by *loadSchema()* or *loadSchemaFile()*.

The return value is a tuple containing the configuration object and a composite handler that, when called with a name-to-handler mapping, calls all the handlers for the configuration. The *overrides* argument is the same as for the *loadConfig()* function.

See also:

ConfigLoader, *BaseLoader.loadFile()*, *ExtendedConfigLoader.addOption()*

ZConfig.**loadSchema**(*url*)

Load a schema definition from the URL *url*.

url may be a URL, absolute pathname, or relative pathname. Fragment identifiers are not supported.

The resulting schema object can be passed to *loadConfig()* or *loadConfigFile()*. The schema object may be used as many times as needed.

See also:

SchemaLoader, *BaseLoader.loadURL()*

ZConfig.**loadSchemaFile**(*file*, *url=None*)

Load a schema definition from the open file object *file*.

If *url* is given and not None, it should be the URL of resource represented by *file*. If *url* is omitted or None, a URL may be computed from the name attribute of *file*, if present. The resulting schema object can be passed to *loadConfig()* or *loadConfigFile()*. The schema object may be used as many times as needed.

See also:

SchemaLoader, *BaseLoader.loadFile()*

4.1.2 Exceptions

The following exceptions are defined by this package:

exception ZConfig.**ConfigurationError**

Bases: Exception

Base class for exceptions specific to the *ZConfig* package.

All instances provide a message attribute that describes the specific error, and a url attribute that gives the URL of the resource the error was located in, or None.

exception ZConfig.**ConfigurationSyntaxError**

Exception raised when a configuration source does not conform to the allowed syntax.

In addition to the message and url attributes, exceptions of this type offer the lineno attribute, which provides the line number at which the error was detected.

exception ZConfig.**DataConversionError**

Bases: *ConfigurationError*, ValueError

Raised when a data type conversion fails with ValueError.

This exception is a subclass of both *ConfigurationError* and ValueError. The str() of the exception provides the explanation from the original ValueError, and the line number and URL of the value which provoked the error. The following additional attributes are provided:

colno
column number at which the value starts, or None

exception
the original ValueError instance

lineno
line number on which the value starts

message
str() returned by the original ValueError

value
original value passed to the conversion function

url
URL of the resource providing the value text

exception ZConfig.**SchemaError**

Raised when a schema contains an error.

This exception type provides the attributes `url`, `lineno`, and `colno`, which provide the source URL, the line number, and the column number at which the error was detected. These attributes may be `None` in some cases.

exception ZConfig.**SchemaResourceError**

Bases: *SchemaError*

Raised when there's an error locating a resource required by the schema.

Instances of this exception class add the attributes `filename`, `package`, and `path`, which hold the filename searched for within the package being loaded, the name of the package, and the `__path__` attribute of the package itself (or `None` if it isn't a package or could not be imported).

exception ZConfig.**SubstitutionReplacementError**

Bases: *ConfigurationSyntaxError*, LookupError

Raised when the source text contains references to names which are not defined in *mapping*.

The attributes `source` and `name` provide the complete source text and the name (converted to lower case) for which no replacement is defined.

exception ZConfig.**SubstitutionSyntaxError**

Raised when interpolation source text contains syntactical errors.

4.1.3 Basic Usage

The simplest use of *ZConfig* is to load a configuration based on a schema stored in a file. This example loads a configuration file specified on the command line using a schema in the same directory as the script:

```
import os
import sys
import ZConfig

try:
    myfile = __file__
except NameError:
    myfile = os.path.realpath(sys.argv[0])

mydir = os.path.dirname(myfile)

schema = ZConfig.loadSchema(os.path.join(mydir, 'schema.xml'))
conf, handler = ZConfig.loadConfig(schema, sys.argv[1])
```

If the schema file contained this schema:

```
<schema>
  <key name='server' required='yes'/>
  <key name='attempts' datatype='integer' default='5'/>
</schema>
```

and the file specified on the command line contained this text:

```
# sample configuration

server www.example.com
```

then the configuration object `conf` loaded above would have two attributes, *server* with the value `'www.example.com'` and *attempts* with the value `5`.

4.2 ZConfig.datatypes — Default data type registry

Default implementation of a data type registry

This module provides the implementation of the default data type registry and all the standard data types supported by *ZConfig*. A number of convenience classes are also provided to assist in the creation of additional data types.

A "data type registry" is an object that provides conversion functions for data types. The interface for a *registry* is fairly simple.

A "conversion function" is any callable object that accepts a single argument and returns a suitable value, or raises an exception if the input value is not acceptable. `ValueError` is the preferred exception for disallowed inputs, but any other exception will be properly propagated.

class ZConfig.datatypes.**Registry**(*stock=None*)

Implementation of a simple type registry.

If given, *stock* should be a mapping which defines the "built-in" data types for the registry; if omitted or `None`, the standard set of data types is used (see *Standard ZConfig Datatypes*).

get(*name*)

Return the type conversion routine for *name*.

If the conversion function cannot be found, an (unspecified) exception is raised. If the name is not provided in the stock set of data types by this registry and has not otherwise been registered, this method uses the *search()* method to load the conversion function. This is the only method the rest of *ZConfig* requires.

register(*name*, *conversion*)

Register the data type name *name* to use the conversion function *conversion*.

If *name* is already registered or provided as a stock data type, `ValueError` is raised (this includes the case when *name* was found using the *search()* method).

search(*name*)

This is a helper method for the default implementation of the *get()* method.

If *name* is a Python dotted-name, this method loads the value for the name by dynamically importing the containing module and extracting the value of the name. The name must refer to a usable conversion function.

The following classes are provided to define conversion functions:

class ZConfig.datatypes.**MemoizedConversion**(*conversion*)

Simple memoization for potentially expensive conversions.

This conversion helper caches each successful conversion for re-use at a later time; failed conversions are not cached in any way, since it is difficult to raise a meaningful exception providing information about the specific failure.

class ZConfig.datatypes.**RangeCheckedConversion**(*conversion*, *min=None*, *max=None*)

Conversion helper that performs range checks on the result of another conversion.

Values passed to instances of this conversion are converted using *conversion* and then range checked. *min* and *max*, if given and not `None`, are the inclusive endpoints of the allowed range. Values returned by *conversion* which lay outside the range described by *min* and *max* cause `ValueError` to be raised.

class ZConfig.datatypes.**RegularExpressionConversion**(*regex*)

Conversion that checks that the input matches the regular expression *regex*.

If it matches, returns the input, otherwise raises `ValueError`.

4.3 ZConfig.loader — Resource loading support

This module provides some helper classes used by the primary APIs exported by the *ZConfig* package. These classes may be useful for some applications, especially applications that want to use a non-default data type registry.

class ZConfig.loader.**Resource**(*file*, *url*)

Object that allows an open file object and a URL to be bound together to ease handling.

Instances have the attributes `file` and `url`, which store the constructor arguments. These objects also have a `close()` method which will call `close()` on *file*, then set the `file` attribute to `None` and the `closed` attribute to `True`. Using this object as a context manager also closes the file.

All other attributes are delegated to *file*.

class ZConfig.loader.**ConfigLoader**(*schema*)

Bases: *BaseLoader*

Loader for configuration files.

Each configuration file must conform to the schema *schema*. The `load*()` methods return a tuple consisting of the configuration object and a composite handler.

class ZConfig.loader.**SchemaLoader**(*registry=None*)

Bases: *BaseLoader*

Loader that loads schema instances.

All schema loaded by a *SchemaLoader* will use the same data type registry. If *registry* is provided and not `None`, it will be used, otherwise an instance of *ZConfig.datatypes.Registry* will be used.

4.3.1 Loader Objects

Loader objects provide a general public interface, an interface which subclasses must implement, and some utility methods.

class ZConfig.loader.**BaseLoader**

Base class for loader objects.

This should not be instantiated directly, as the *loadResource()* method must be overridden for the instance to be used via the public API.

The following methods provide the public interface:

BaseLoader.**loadURL**(*url*)

Open and load a resource specified by the URL *url*.

This method uses the *loadResource()* method to perform the actual load, and returns whatever that method returns.

BaseLoader.**loadFile**(*file*, *url=None*)

Load from an open file object, *file*.

If given and not None, *url* should be the URL of the resource represented by *file*. If omitted or *None*, the name attribute of *file* is used to compute a file: URL, if present.

This method uses the *loadResource()* method to perform the actual load, and returns whatever that method returns.

The following method must be overridden by subclasses:

abstract BaseLoader.**loadResource**(*resource*)

Abstract method.

Subclasses of *BaseLoader* must implement this method to actually load the resource and return the appropriate application-level object.

The following methods can be used as utilities:

BaseLoader.**isPath**(*s*)

Return true if *s* should be considered a filesystem path rather than a URL.

BaseLoader.**normalizeURL**(*url*)

Return a URL for *url*

If *url* refers to an existing file, the corresponding file: URL is returned. Otherwise *url* is checked for sanity: if it does not have a schema, ValueError is raised, and if it does have a fragment identifier, *ConfigurationError* is raised.

This uses *isPath()* to determine whether *url* is a URL of a filesystem path.

BaseLoader.**openResource**(*url*)

Returns a resource object that represents the URL *url*.

The URL is opened using the urllib.request.urlopen() function, and the returned resource object is created using *createResource()*. If the URL cannot be opened, *ConfigurationError* is raised.

BaseLoader.**createResource**(*file*, *url*)

Returns a resource object for an open file and URL, given as *file* and *url*, respectively.

This may be overridden by a subclass if an alternate resource implementation is desired.

4.4 ZConfig.substitution — String substitution

Shell-style string substitution helper.

This module provides a basic substitution facility similar to that found in the Bourne shell (sh on most UNIX platforms).

The replacements supported by this module include:

Source	Replacement	Notes
`$$`	`$`	(1)
`$name`	The result of looking up *name*	(2)
`${name}`	The result of looking up *name*	
`$(name)`	The result of looking up *name* in the environment	(3)

Notes:

1. This is different from the Bourne shell, which uses `\$` to generate a `$` in the result text. This difference avoids having as many special characters in the syntax.
2. Any character which immediately follows *name* may not be a valid character in a name.
3. This is not Bourne shell style.

In each case, *name* is a non-empty sequence of alphanumeric and underscore characters not starting with a digit. If there is not a replacement for *name*, the exception *SubstitutionReplacementError* is raised. Note that the lookup is expected to be case-insensitive; this module will always use a lower-case version of the name to perform the query.

This module provides these functions:

`ZConfig.substitution.`**`substitute`**`(`*s*, *mapping*`)`

Substitute values from *mapping* into *s*.

mapping can be a `dict` or any type that supports the `get()` method of the mapping protocol. Replacement values are copied into the result without further interpretation. Raises *SubstitutionSyntaxError* if there are malformed constructs in *s*.

`ZConfig.substitution.`**`isname`**`(`*s*`)`

Returns `True` if *s* is a valid name for a substitution text, otherwise returns `False`.

4.4.1 Examples

```
>>> from ZConfig.substitution import substitute
>>> d = {'name': 'value',
...      'top': '$middle',
...      'middle' : 'bottom'}
>>>
>>> substitute('$name', d)
'value'
>>> substitute('$top', d)
'$middle'
>>> import os
>>> os.environ['from_environment'] = 'From environment.'
>>> substitute('$(from_einvironment)', d)
'From environment.'
```

4.5 ZConfig.cmdline — Command-line override support

Support for command-line overrides for configuration settings.

This module exports an extended version of the *ConfigLoader* class from the *ZConfig.loader* module. This provides support for overriding specific settings from the configuration file from the command line, without requiring the application to provide specific options for everything the configuration file can include.

Each setting is given by a value specifier string, as described by *ExtendedConfigLoader.addOption()*.

class ZConfig.cmdline.**ExtendedConfigLoader**(*schema*)

Bases: *ConfigLoader*

A *ConfigLoader* subclass that adds support for command-line overrides.

The following additional method is provided, and is the only way to provide position information to associate with command-line parameters:

ExtendedConfigLoader.**addOption**(*spec*, *pos=None*)

Add a single value to the list of overridden values.

The *spec* argument is a value specifier string of the form `optionpath=value`. For example:

```
some/path/to/key=value
```

The *optionpath* specifies the "full path" to the configuration setting: it can contain a sequence of names, separated by `/` characters. Each name before the last names a section from the configuration file, and the last name corresponds to a key within the section identified by the leading section names. If *optionpath* contains only one name, it identifies a key in the top-level schema. *value* is a string that will be treated just like a value in the configuration file.

A source position for the specifier may be given as *pos*. If *pos* is specified and not `None`, it must be a sequence of three values. The first is the URL of the source (or some other identifying string). The second and third are the line number and column of the setting. These position information is only used to construct a *DataConversionError* when data conversion fails.

CHAPTER

FIVE

ZCONFIG TOOLING

ZConfig ships with some tools that can be helpful to anyone either writing configurations or writing programs that read configurations.

5.1 Schema and Configuration Validation

When ZConfig is installed, it installs a program called `zconfig` that can validate both schemas and configurations written against those schemas:

```
usage: zconfig [-h] -s FILE [file ...]

Script to check validity of a configuration file

positional arguments:
  file                  Optional configuration file to check

options:
  -h, --help            show this help message and exit
  -s FILE, --schema FILE
                        use the schema in FILE (can be a URL)

Each file named on the command line is checked for syntactical errors and
schema conformance. The schema must be specified. If no files are specified
and standard input is not a TTY, standard in is treated as a configuration
file. Specifying a schema and no configuration files causes the schema to be
checked.
```

5.2 Documenting Schemas

ZConfig also installs a tool called `zconfig_schema2html` that can print schemas in a simple HTML format.

Hint: To document components in reStructuredText, e.g., with Sphinx, see *Documenting Components*.

```
usage: zconfig_schema2html [-h] [--out OUT] [--package]
                           [--package-file PACKAGE_FILE]
                           [--members [MEMBERS ...]] [--format {html,xml}]
```

(continues on next page)

(continued from previous page)

```
                            [SCHEMA-OR-PACKAGE]

Print an HTML version of a schema

positional arguments:
  [SCHEMA-OR-PACKAGE]   The schema to print. By default, a file. Optionally, a
                        Python package. If not given, defaults to reading a
                        schema file from stdin

options:
  -h, --help            show this help message and exit
  --out OUT, -o OUT     Write the schema to this file; if not given, write to
                        stdout
  --package             The SCHEMA-OR-PACKAGE argument indicates a Python
                        package instead of a file. The component.xml (by
                        default) from the package will be read.
  --package-file PACKAGE_FILE
                        When PACKAGE is given, this can specify the file
                        inside it to load.
  --members [MEMBERS ...]
                        Only output sections and types in this list (and
                        reachable from it).
  --format {html,xml}   The output format to produce.
```

CHAPTER

SIX

CHANGE HISTORY FOR ZCONFIG

6.1 4.1 (2024-05-03)

- Add support for Python 3.12.

6.2 4.0 (2023-05-05)

- Drop support for Python 2.7, 3.5, 3.6.

6.3 3.6.1 (2022-12-06)

- Add support for Python 3.11.
- Drop support for Python 3.4.

6.4 3.6.0 (2021-05-19)

- Added support for Python 3.8, 3.9 and 3.10. This primarily involves avoiding the new-in-3.8 validation of the format string when using the 'safe-template' format style, since that's not supported in the Python standard library.
- Added `ZConfig.pygments` module containing a lexer compatible with the `pygments` library. Made discoverable via an entry point; use **zconfig** as the highlight language for `code-block` directives in Sphinx documents.

6.5 3.5.0 (2019-06-24)

- Add support for documenting schema files contained in packages to the Sphinx extension. See issue 59.

6.6 3.4.0 (2019-01-02)

Many changes have been made in the support for logging configurations:

- The log handler section types defined by the `ZConfig.components.logger` package support additional, optional parameters:

  `style`
  : Used to configure alternate format styles as found in the Python 3 standard library. Four `style` values are supported: `classic` (the default), `format` (equivalent to `style='{'` in Python 3), `template` (equivalent to `style='$'`), and `safe-template` (similar to `template`, but using the `string.Template` method `safe_substitute` method). A best-effort implementation is provided for Python 2.

  `arbitrary-fields`
  : A Boolean defauting to `False` for backward compatibility, allows arbitrary replacement field names to be accepted in the format string (regardless of the `style` setting). This supports applications where log records are known to be generated with additional string or numeric fields, at least for some loggers. (An exception is still raised at format time if the additional fields are not provided, unless the `style` value `safe-template` is used.)

- The `logfile` section type defined by the `ZConfig.components.logger` package supports the optional `delay` and `encoding` parameters. These can only be used for regular files, not the special `STDOUT` and `STDERR` streams.
- More validation on the parameters to the `logfile` and `email-notifier` sections is performed early (at the construction of the factory, rather than at creation of the `logging` handler). This allows more checking of parameter combinations before any log files are opened.
- The `ZConfig.components.logger.handlers.log_format` data type function now supports formats that include numeric formatting for `levelno`, and accept `funcName` as a valid log record field (added in Python 2.6 and 3.1).

6.7 3.3.0 (2018-10-04)

- Drop support for Python 3.3.
- Add support for Python 3.7.
- Drop support for 'python setup.py test'. See issue 38.
- Add support for `example` in `section` and `multisection`, and include those examples in generated documentation. See https://github.com/zopefoundation/ZConfig/pull/5.
- Fix configuration loaders to decode byte data using UTF-8 instead of the default encoding (usually ASCII). See issue 37.

6.8 3.2.0 (2017-06-22)

- Drop support for Python 2.6 and 3.2 and add support for Python 3.6.
- Run tests with pypy and pypy3 as well.
- Host docs at https://zconfig.readthedocs.io
- BaseLoader is now an abstract class that cannot be instantiated.
- Allow `nan`, `inf` and `-inf` values for floats in configurations. See https://github.com/zopefoundation/ZConfig/issues/16.

- Scripts `zconfig` (for schema validation) and `zconfig_schema2html` are ported to Python 3.
- A new `ZConfig.sphinx` Sphinx extension facilitates automatically documenting ZConfig components using their description and examples in Sphinx documentation. See https://github.com/zopefoundation/ZConfig/pull/25.
- Simplify internal schema processing of max and min occurrence values. See https://github.com/zopefoundation/ZConfig/issues/15.
- Almost all uses of `type` as a parameter name have been replaced with `type_` to avoid shadowing a builtin. These were typically not public APIs and weren't expected to be called with keyword arguments so there should not be any user-visible changes. See https://github.com/zopefoundation/ZConfig/issues/17

6.9 3.1.0 (2015-10-17)

- Add ability to do variable substitution from environment variables using $() syntax.

6.10 3.0.4 (2014-03-20)

- Added Python 3.4 support.

6.11 3.0.3 (2013-03-02)

- Added Python 3.2 support.

6.12 3.0.2 (2013-02-14)

- Fixed ResourceWarning in BaseLoader.openResource().

6.13 3.0.1 (2013-02-13)

- Removed an accidentally left *pdb* statement from the code.
- Fix a bug in Python 3 with the custom string *repr()* function.

6.14 3.0.0 (2013-02-13)

- Added Python 3.3 support.
- Dropped Python 2.4 and 2.5 support.

6.15 2.9.3 (2012-06-25)

- Fixed: port values of 0 weren't allowed. Port 0 is used to request an ephemeral port.

6.16 2.9.2 (2012-02-11)

- Adjust test classes to avoid base classes being considered separate test cases by (at least) the "nose" test runner.

6.17 2.9.1 (2012-02-11)

- Make FileHandler.reopen thread safe.

6.18 2.9.0 (2011-03-22)

- Allow identical redefinition of `%define` names.
- Added support for IPv6 addresses.

6.19 2.8.0 (2010-04-13)

- Fix relative path recognition. https://bugs.launchpad.net/zconfig/+bug/405687
- Added SMTP authentication support for email logger on Python 2.6.

6.20 2.7.1 (2009-06-13)

- Improved documentation
- Fixed tests failures on windows.

6.21 2.7.0 (2009-06-11)

- Added a convenience function, `ZConfig.configureLoggers(text)` for configuring loggers.
- Relaxed the requirement for a logger name in logger sections, allowing the logger section to be used for both root and non-root loggers.

6.22 2.6.1 (2008-12-05)

- Fixed support for schema descriptions that override descriptions from a base schema. If multiple base schema provide descriptions but the derived schema does not, the first base mentioned that provides a description wins. https://bugs.launchpad.net/zconfig/+bug/259475
- Fixed compatibility bug with Python 2.5.0.
- No longer trigger deprecation warnings under Python 2.6.

6.23 2.6.0 (2008-09-03)

- Added support for file rotation by time by specifying when and interval, rather than max-size, for log files.
- Removed dependency on setuptools from the setup.py.

6.24 2.5.1 (2007-12-24)

- Made it possible to run unit tests via 'python setup.py test' (requires setuptools on sys.path).
- Added better error messages to test failure assertions.

6.25 2.5 (2007-08-31)

A note on the version number:

Information discovered in the revision control system suggests that some past revision has been called "2.4", though it is not clear that any actual release was made with that version number. We're going to skip revision 2.4 entirely to avoid potential issues with anyone using something claiming to be ZConfig 2.4, and go straight to version 2.5.

- Add support for importing schema components from ZIP archives (including eggs).
- Added a 'formatter' configuration option in the logging handler sections to allow specifying a constructor for the formatter.
- Documented the package: URL scheme that can be used in extending schema.
- Added support for reopening all log files opened via configurations using the ZConfig.components.logger package. For Zope, this is usable via the `zc.signalhandler` package. `zc.signalhandler` is not required for ZConfig.
- Added support for rotating log files internally by size.
- Added a minimal implementation of schema-less parsing; this is mostly intended for applications that want to read several fragments of ZConfig configuration files and assemble a combined configuration. Used in some `zc.buildout` recipes.
- Converted to using `zc.buildout` and the standard test runner from `zope.testing`.
- Added more tests.

6.26 2.3.1 (2005-08-21)

- Isolated some of the case-normalization code so it will at least be easier to override. This remains non-trivial.

6.27 2.3 (2005-05-18)

- Added "inet-binding-address" and "inet-connection-address" to the set of standard datatypes. These are similar to the "inet-address" type, but the default hostname is more sensible. The datatype used should reflect how the value will be used.
- Alternate rotating logfile handler for Windows, to avoid platform limitations on renaming open files. Contributed by Sidnei da Silva.
- For <section> and <multisection>, if the name attribute is omitted, assume name="*", since this is what is used most often.

6.28 2.2 (2004-04-21)

- More documentation has been written.
- Added a timedelta datatype function; the input is the same as for the time-interval datatype, but the resulting value is a datetime.timedelta object.
- Make sure keys specified as attributes of the <default> element are converted by the appropriate key type, and are re-checked for derived sections.
- Refactored the ZConfig.components.logger schema components so that a schema can import just one of the "eventlog" or "logger" sections if desired. This can be helpful to avoid naming conflicts.
- Added a reopen() method to the logger factories.
- Always use an absolute pathname when opening a FileHandler.
- A fix to the logger 'format' key to allow the %(process)d expansion variable that the logging package supports.
- A new timedelta built-in datatype was added. Similar to time-interval except that it returns a datetime.timedelta object instead.

6.29 2.1 (2004-04-12)

- Removed compatibility with Python 2.1 and 2.2.
- Schema components must really be in Python packages; the directory search has been modified to perform an import to locate the package rather than incorrectly implementing the search algorithm.
- The default objects use for section values now provide a method getSectionAttributes(); this returns a list of all the attributes of the section object which store configuration-defined data (including information derived from the schema).
- Default information can now be included in a schema for <key name="+"> and <multikey name="+"> by using <default key="...">.
- More documentation has been added to discuss schema extension.
- Support for a Unicode-free Python has been fixed.

- Derived section types now inherit the datatype of the base type if no datatype is identified explicitly.
- Derived section types can now override the keytype instead of always inheriting from their base type.
- <import package='...'/> makes use of the current prefix if the package name begins witha dot.
- Added two standard datatypes: dotted-name and dotted-suffix.
- Added two standard schema components: ZConfig.components.basic and ZConfig.components.logger.

6.30 2.0 (2003-10-27)

- Configurations can import additional schema components using a new "%import" directive; this can be used to integrate 3rd-party components into an application.
- Schemas may be extended using a new "extends" attribute on the <schema> element.
- Better error messages when elements in a schema definition are improperly nested.
- The "zconfig" script can now simply verify that a schema definition is valid, if that's all that's needed.

6.31 1.0 (2003-03-25)

- Initial release.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

Z

INDEX