# ZConfig Package Reference

Zope Corporation

February 20, 2003

Lafayette Technology Center
513 Prince Edward Street
Fredericksburg, VA 22401
http://www.zope.com/

**Abstract**

This document describes the syntax and API used in configuration files for components of a Zope installation written by Zope Corporation. This configuration mechanism is itself configured using a schema specification written in XML.

**Warning:** ZConfig has changed a great deal since this document was initially written, and parts of this have not yet been updated, though portions have been. Please be patient as the documentation catches up.

## Contents

# 1 Introduction

Zope uses a common syntax and API for configuration files designed for software components written by Zope Corporation. Third-party software which is also part of a Zope installation may use a different syntax, though any software is welcome to use the syntax used by Zope Corporation. Any software written in Python is free to use the `ZConfig` software to load such configuration files in order to ensure compatibility. This software is covered by the Zope Public License, version 2.0.

The `ZConfig` package has been tested with Python 2.1 and 2.2. Python 2.0 is not supported. `ZConfig` only relies on the Python standard library.

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.

# 2 Configuration Syntax

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. 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*] />

## 2.1 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. All resources being parsed as part of a

configuration share a single namespace for defined names. This means that resources which may be included more than once should not define any 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
```

# 3 Writing Configuration Schema

XXX to be written

`ZConfig` schema are written as XML documents.

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.

## 3.1 Schema Elements

XXX need to discuss notation

The following elements are used to describe a schema:

<**schema**>
    `description?, metadefault?, example?, import*, (sectiontype | abstract-type)*, (section | key | multisection | multikey)*`
</**schema**>
    Document element for a `ZConfig` schema.

    **`datatype`** (**basic-key** or **dotted-name**)
        The data type converter which will be applied to the value of this section.

    **`handler`** (**basic-key**)

    **`keytype`** (**basic-key**)

    **`prefix`** (**dotted-name**)

<**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.

    **`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?, (section | key)*
</**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.

    **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 are sections contained within the `sectiontype` element.

    **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**)

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

    **prefix** (**dotted-name**)

<**import**>
    EMPTY
</**import**>
    Import a schema component. Exactly one of the two possible attributes must be specified.

    **package** (**dotted-name**)
        Python-package style name that identifies a directory found on `sys.path` containing a schema component in a file named 'component.xml'. Dots in the value are converted to directory separators.

    **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?`
</**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.

**`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`** (**dotted-name**)

**`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 `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.

**`handler`** (**dotted-name**)

**`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.

<**section**>
    description?
</**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`** (**dotted-name**)

**`name`** (**basic-key**)
: The name of the section, 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. 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?
</**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`** (**dotted-name**)

**`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 `*`, 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.

# 4 Schema Components

XXX need more explanation

ZConfig supports extensible schema components that can be provided by disparate components, and allows them to be knit together into concrete schema for applications. Neither components nor extensions can add additional keys or sections in the application schema.

A schema *component* is allowed to define new abstract and section types. It is not allowed to extend application types or include additional types in application-provided abstract types. Components are identified using a dotted-name, similar to a Python module name. For example, one component may be zodb.storage.

A schema component *extension* is allowed to define new abstract and section types, extend types provided by the component it extends, and include new section types in abstract types provided by the component. The expected usage is that extensions will provide one or more concrete types that implement abstract types defined by the component.

A library of schema components is stored as a directory tree, where each component is located in a directory within the tree. That directory must contain a file named 'component.xml' which defines the types provided by that component; it must have a component element as the document element. Extensions to a component are stored in immediate subdirectories; a file 'extension.xml' provides the extension types. Extensions must have an extension element as the document element.

## 4.1 Schema Component Elements

The following elements are used as the document elements of schema components and schema component extensions.

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

The top-level element for schema components.

**prefix** (**dotted-name**)

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

The top-level element for schema component extensions.

**prefix** (**dotted-name**)

# 5 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`'

**constructor**
Parse value in the form '`fn('1', '2', kw1='a', kw2='b')`' into a 3-tuple where the first element is the string `'fn'`, the 2nd element is the list `['1', '2']`, and the 3rd element is the dictionary `{'kw1': 'a', 'kw2': 'b'}`. This is useful when representing a Python-style constructor as a value. Python syntax rules are enforced, but only constants are allowed as positional and keyword arguments. The 3-tuple is returned.

**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, an empty string 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 `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. Hostnames are converted to lower case.

**key-value**
Parse a value in the form `'A B'` into the list `['A', 'B']`. Returns the list.

**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 (`AF_INET` or `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 key values in configuration files.

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

# 6 `ZConfig` — Basic configuration support

The main `ZConfig` package exports these convenience functions:

**`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*. 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.

**`loadConfigFile`**(*schema, file*[, *url*[, *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.

**`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.

**`loadSchemaFile`**(*file*[, *url*])
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.

The following exceptions are defined by this package:

**exception `ConfigurationError`**
Base class for exceptions specific to the `ZConfig` package. All instances provide a `message` attribute that describes the specific error.

**exception `ConfigurationSyntaxError`**
Exception raised when a configuration source does not conform to the allowed syntax. In addition to the `message` attribute, exceptions of this type offer the `url` and `lineno` attributes, which provide the URL and line number at which the error was detected.

**exception `ConfigurationTypeError`**

**exception `ConfigurationMissingSectionError`**
Raised when a requested named section is not available.

**exception `ConfigurationConflictingSectionError`**
Raised when a request for a section cannot be fulfilled without ambiguity.

**exception `DataConversionError`**
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:

| Attribute | Value |
|---|---|
| `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 `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 `SubstitutionReplacementError`**
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 `SubstitutionSyntaxError`**
Raised when the source text contains syntactical errors.

## 6.1 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:
    # really should follow symlinks here:
    myfile = sys.argv[0]

mydir = os.path.dirname(os.path.abspath(myfile))

schema = ZConfig.loadSchema(os.path.join(mydir, 'schema.xml'))
conf = 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:

| Attribute | Value |
|---|---|
| `server` | `'www.example.com'` |
| `attempts` | `5` |

# 7 `ZConfig.datatypes` — Default data type registry

The `ZConfig.datatypes` 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 *datatype 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 propogated.

**class `Registry`**([*stock*])

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 section 5, "Standard `ZConfig` Datatypes").

`Registry` objects have the following methods:

**`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 `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 `RangeCheckedConversion`**(*conversion*[, *min*[, *max*]])
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 `RegularExpressionConversion`**(*regex*)
Conversion that checks that the input matches the regular expression *regex*. If it matches, returns the input, otherwise raises `ValueError`.

# 8 `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 `Resource`**(*file, url*[, *fragment*])
Object that allows an open file object and a URL to be bound together to ease handling. Instances have the attributes `file`, `url`, and `fragment` 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` to `True`.

**class `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.

**class `ConfigLoader`**(*schema*)
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 `SchemaLoader`**([*registry*])
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.

## 8.1 Loader Objects

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

The following methods provide the public interface:

**`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.

**`loadFile`**(*file*[, *url*])
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:

**`loadResource`**(*resource*)
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:

**`normalizeURL`**(*url-or-path*)
Return a URL for *url-or-path*. If *url-or-path* refers to an existing file, the corresponding `file:` URL is returned. Otherwise *url-or-path* 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.

**`openResource`**(*url*)
Returns a resource object that represents the URL *url*. The URL is opened using the `urllib2.urlopen()` function, and the returned resource object is created using `createResource()`.

**`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.

# 9 `ZConfig.substitution` — String substitution

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* | |

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.

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:

**`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*.

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

## 9.1 Examples

```
>>> from ZConfig.substitution import substitute
>>> d = {'name': 'value',
...      'top': '$middle',
...      'middle' : 'bottom'}
>>>
>>> substitute('$name', d)
'value'
>>> substitute('$top', d)
'$middle'
```

# A Schema Document Type Definition

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

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

  This software is subject to the provisions of the Zope Public License,
  Version 2.0 (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.
  ############################################################################
  -->

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

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

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

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

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

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

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

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

<!ELEMENT section (description?)>
<!ATTLIST section
          name       NMTOKEN  #REQUIRED
          attribute  NMTOKEN  #IMPLIED
          type       NMTOKEN  #REQUIRED
          handler    NMTOKEN  #IMPLIED
          minOccurs  NMTOKEN  #IMPLIED
```

```
            maxOccurs  NMTOKEN  #IMPLIED>

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

# B `ZConfig.Context` — Application context (obsolete)

**Warning:** This module is provided for backward compatibility. It may be removed at some point in the future. The configuration objects returned by methods of the `Context` object described here are very different from the schema-based configuration objects.

The `ZConfig` package uses the idea of an *application context* to consolidate the connections between the different components of the package. Most applications should be able to use the context implementation provided in this module.

For applications that need to change the way their configuration data is handled, the best way to do it is to provide an alternate application context. The default implementation is designed to be subclassed, so this should not prove to be difficult.

**class `Context`()**
: Constructs an instance of the default application context. This is implemented as an object to allow applications to adjust the way components are created and how they are knit together. This implementation is designed to be used once and discarded; changing this assumption in a subclass would probably lead to a complete replacement of the class.

The context object offers two methods that are used to load a configuration. Exactly one of these methods should be called, and it should be called only once:

**`loadURL`**(*url*)
: Load and return a configuration object from a resource. The resource is identified by a URL or path given as *url*. Fragment identifiers are not supported.

**`loadFile`**(*file*[, *url*])
: 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.

The following methods are defined to be individually overridden by subclasses; this should suffice for most context specialization.

**`createNestedSection`**(*parent, type, name, delegatename*)
: Create a new section that represents a child of the section given by *parent*. *type* is the type that should be given to the new section and should always be a string. *name* should be the name of the section, and should be a string or `None`. *delegatename* should also be a string or `None`; if not `None`, this will be the name of the section eventually passed to the `setDelegate()` method of the returned section. The returned section should be conform to the interface of the `Configuration` class (see the `ZConfig.Config` module's documentation for more information on this interface).

**`createToplevelSection`**(*url*)
: Create a new section that represents a section loaded and returned by the `loadURL()` method of the context object. The returned section should be conform to the interface of the `ImportingConfiguration` class (see the `ZConfig.Config` module's documentation for more information on this interface). *url* is the resource that will be loaded into the new section. Since the new section represents the top level of an external resource,

it's `type` and `name` attributes should be `None`.

**`getDelegateType`**(*type*)
Return the type of sections to which sections of type *type* may delegate to, or `None` if they are not allowed to do so.

**`parse`**(*resource, section*)
This method allows subclasses to replace the resource parser. *resource* is an object that represents a configuration source; it has two attributes, `file` and `url`. The `file` attribute is a file object which provides the content of the resource, and `url` is the URL from which the resource is being loaded. *section* is the section object into which the contents of the resources should be loaded. The default implementation implements the configuration language described in section 2. Providing an alternate parser is most easily done by overriding this method and calling the parser support methods of the context object from the new parser, though different strategies are possible.

The following methods are provided to make it easy for parsers to support common semantics for the `%include` statement, if that is defined for the syntax implemented by the alternate parser.

**`includeConfiguration`**(*parent, url*)

**`startSection`**(*parent, type, name, delegatename*)

**`endSection`**(*parent, type, name, delegatename, section*)

# C `ZConfig.Config` — Section objects (obsolete)

**Warning:** This module is provided for backward compatibility. It may be removed at some point in the future. It should really be considered an implementation detail of the configuration objects returned by methods of the `Context` object defined in the `ZConfig.Context` module.

The `ZConfig.Config` module provides an implementation of the standard key-value section for configurations loaded by the `ZConfig.Context` module.

**class `Configuration`**(*type, name, url*)
A typed section with an optional name. The type is given by the *type* argument, and the URL the configuration is loaded from is given by *url*. Both *type* and *url* must be strings. The optional name of the section is given by *name*; if there is no name, *name* should be `None`.

`Configuration` objects provide the following attributes and methods to retrieve information from the section:

**`container`**
The containing section of this section, or `None`.

**`delegate`**
The `Configuration` object to which lookups are delegated when they cannot be satisfied directly. If there is no such section, this will be `None`.

**`get`**(*key*[, *default*])
Returns the value for *key* as a string; a value from the delegate section is used if needed. If there is no value for *key*, returns *default*.

**`getbool`**(*key*[, *default*])
Returns the value for *key* as a `bool`. If there is no value for *key*, returns *default*. Conversions to `bool` are case-insensitive; the strings `true`, `yes`, and `on` cause `True` to be returned; the strings `false`, `no`, and `off` generate `False`. All other strings cause `ValueError` to be raised.

**`getfloat`**(*key*[, *default*[, *min*[, *max*]]])
Return the value for *key* as a float. If there is no value for *key*, returns *default*. If the value cannot be converted to a float, `ValueError` is raised. If *min* is given and the value is less than *min*, or if *max* is given and the value is greater than *max*, `ValueError` is raised. No range checking is performed if neither *min* nor *max* is given.

**getint**(*key*[, *default*[, *min*[, *max*]]])
Return the value for *key* as an integer. If there is no value for *key*, returns *default*. If the value cannot be converted to an integer, `ValueError` is raised. If *min* is given and the value is less than *min*, or if *max* is given and the value is greater than *max*, `ValueError` is raised. No range checking is performed if neither *min* nor *max* is given.

**getlist**(*key*[, *default*])
Return the value for *key*, converted to a list. List items are separated by whitespace.

**has_key**(*key*)
Return `True` if *key* has an associated value, otherwise returns `False`.

**items**()
Return a list of key-value pairs from this section, including any available from the delegate section.

**keys**()
Return a list of keys from this section, including any available from the delegate section.

**name**
The name of this section, or `None`.

**type**
The type of this section as a string.

**url**
The URL of the source this section was loaded from.

The following method is used to modify the values defined in a section:

**addValue**(*key, value*)
Add the key *key* with the value *value*. If there is already a value for *key*, `ConfigurationError` is raised.

The following methods are used in retrieving and managing sections:

**addChildSection**(*section*)
Add a section that is a child of the current section.

**addNamedSection**(*section*)
Add a named section to this section's context. This is only used to add sections that are descendants but not children of the current section.

**getChildSections**([*type*])
Returns a sequence of all child sections, in the order in which they were added. If *type* is omitted or `None`, all sections are returned; otherwise only sections of the specified type are included. The delegate is never consulted by this method.

**getSection**(*type*[, *name*])
Returns a single typed section. The type of the retrieved section is given by *type*. If *name* is given and not `None`, the name of the section must match *name*. If there is no section matching in both name and type, `ConfigurationMissingSectionError` is raised. If *name* is not given or is `None`, there must be exactly one child section of type *type*; that section is returned. If there is more than one section of type *type*, `ConfigurationConflictingSectionError` is raised. If there is no matching section and a delegate is available, it's `getSection()` method is called to provide the return value, otherwise `None` is returned.

Delegation is supported by one additional method:

**setDelegate**(*section*)
Set the delegate section to *section* if not already set. If already set, raises `ConfigurationError`.

This method is called on each section when the configuration is completely loaded. This is called for all sections contained within a section before it is called on the containing section.

**finish**()

Perform any initialization for the section object that needs to occur after the content of the section is loaded and delegation chains have been established. (This method may not have been called for delegates before being called on the delegating section.) The default implementation does nothing.