Contents

1 Links 3
2 Installation 5
3 Introduction 7
4 Features 9
5 Concepts 11
6 Contents: 13
   6.1 Reference .............................. 13
   6.2 Advanced use ................................ 16
   6.3 Goals ......................................... 19
   6.4 Development ................................ 20
   6.5 ChangeLog ................................. 21
7 Indices and tables 25
Python Module Index 27
The `getconf` project provides simple configuration helpers for Python programs.

It provides a simple API to read from various configuration files and environment variables:

```python
import getconf
cfg = getconf.ConfigGetter('myproj',(['/etc/myproj.conf']))
db_host = cfg.get('db.host', 'localhost')
db_port = cfg.getint('db.port', 5432)
```

Beyond this API, getconf aims at unifying configuration setup across development and production systems, respecting the standard procedures in each system:

- Allow userspace configuration on development systems
- Allow multiple different configurations for continuous integration systems
- Use standard configuration space in `/etc` on traditional production servers
- Handle environment-based configuration for cloud-based platforms

`getconf` v1.6 onwards supports 2.7, 3.3, 3.4, 3.5 and is distributed under the two-clause BSD license. The latest version of `getconf` to support Python 2.6 is v1.5.1.
• Package on PyPI: http://pypi.python.org/pypi/getconf/
• Doc on ReadTheDocs: http://readthedocs.org/docs/getconf/
• Source on GitHub: http://github.com/Polyconseil/getconf/
Installation

Intall the package from PyPI, using pip:

```bash
pip install getconf
```

Or from GitHub:

```bash
git clone git://github.com/Polyconseil/getconf
```

`getconf` has no external dependancy beyond Python.
Note: Please refer to the full doc for reference and advanced usage.

All configuration values are accessed through the `getconf.ConfigGetter` object:

```python
import getconf
config = getconf.ConfigGetter('myproj', [’/etc/myproj/settings.ini’, ’./local_settings.ini’])
```

The above line declares:

- Use the `myproj` namespace (explained later; this is mostly used for environment-based configuration, as a prefix for environment variables)

- Look, in turn, at `/etc/myproj/settings.ini` (for production) and `./local_settings.ini` (for development); the latter overriding the former.

Once the `getconf.ConfigGetter` has been configured, it can be used to retrieve settings:

```python
debug = config.getbool('debug', False)
db_host = config.get('db.host', 'localhost')
db_port = config.getint('db.port', 5432)
allowed_hosts = config.getlist('django.allowed_hosts', ['*'])
```

All settings have a type (default is text), and accept a default value. They use namespaces (think ‘sections’) for easier reading.

With the above setup, `getconf` will try to provide `db.host` by inspecting the following options in order (it stops at the first defined value):

- From the environment variable `MYPROJ_DB_HOST`, if defined
- From the host key in the `[db]` section of `./local_settings.ini`
- From the host key in the `[db]` section of `/etc/myproj/settings.ini`
- From the default provided value, `'localhost'`
Features

Env-based configuration files  An extra configuration file/directory/glob can be provided through MYPROJ_CONFIG; it takes precedence over other files.

Default options  An extra dictionary can be provided as ConfigGetter(defaults=some_dict); it is used after configuration files and environment variables.

It should be a dict mapping a section name to a dict of key => value:

```python
>>> config = ConfigGetter('myproj', defaults={'db': {'host': 'localhost'}})
>>> config.get('db.host')
'localhost'
```

Typed getters  getconf can convert options into a few standard types:

```python
config.getbool('db.enabled', False)
config.getint('db.port', 5432)
config.getlist('db.tables')  # Expects a comma-separated list
config.getfloat('db.auto_vacuum_scale_factor', 0.2)
```
getconf relies on a few key concepts:

**namespace** Each ConfigGetter works within a specific namespace (its first argument). Its goal is to avoid mistakes while reading the environment: with ConfigGetter(namespace='myproj'), only environment variables beginning with MYPROJ_ will be read.

It is, however, possible to disable namespacing by using ConfigGetter(namespace=getconf.NO_NAMESPACE).

**Sections** The configuration options for a project often grow quite a lot; to restrict complexity, getconf splits values into sections, similar to Python’s configparser module.

Section are handled differently depending on the actual configuration source:

- section.key is mapped to MYPROJ_SECTION_KEY for environment variables
- section.key is mapped to [section] key = in configuration files
- section.key is mapped to defaults['section']['key'] in the defaults dict.

**Default section** Some settings are actually “globals” for a projet. This is handled by unset section names:

- key is mapped to MYPROJ_KEY for environment variables
- key is mapped to [DEFAULT] key = in configuration files
- key is mapped to defaults['DEFAULT']['key'] in the defaults dict.
Reference

The ConfigGetter class

class getconf.ConfigGetter (namespace, config_files=[config_file_path, ...], defaults={'section': {'key': 'value', ...}, ...})

This class works as a proxy around both os.environ and INI configuration files.

Parameters

• namespace (str) – The namespace for all configuration entry lookups. If an environment variable of <NAMESPACE>_CONFIG is set, the file at that path will be loaded. Pass in the getconf.NO_NAMESPACE special value to load an empty namespace.

• config_files (list) – List of ini-style configuration files to use. Each item may either be the path to a simple file, or to a directory (if the path ends with a ‘/’) or a glob pattern (which will select all the files matching the pattern according to the rules used by the shell). Each directory path will be replaced by the list of its directly contained files, in alphabetical order, excluding those whose name starts with a ‘.’. Provided configuration files are read in the order their name was provided, each overriding the next ones’ values. <NAMESPACE>_CONFIG takes precedence over all config_files contents.

• defaults (dict) – Dictionary of defaults values that are fetch with the lowest priority. The value for ‘section.key’ will be looked up at defaults[‘section’][‘key’].

Warning: When running with an empty namespace (namespace=getconf.NO_NAMESPACE), the environment variables are looked up under <SECTION>_<KEY> instead of <NAMESPACE>_<SECTION>_<KEY>; use this setup with care, since getconf might load variables that weren’t intended for this application.

getstr (key[, default=''])

Retrieve a key from available environments.

Parameters

• key (str) – The name of the field to use.

• default (str) – The default value (string) for the field; optional

Note: The key param accepts two formats:
• 'foo.bar', mapped to section 'foo', key 'bar'
• 'foo', mapped to section '', key 'bar'

This looks, in order, at:
• <NAMESPACE>_<SECTION>_<KEY> if section is set, <NAMESPACE>_<KEY> otherwise
• The <key> entry of the <section> section of the file given in <NAMESPACE>_CONFIG
• The <key> entry of the <section> section of each file given in config_files
• The default value

def getlist(key[, default=()])
Retrieve a key from available configuration sources, and parse it as a list.

Warning: The default value has the same syntax as expected values, e.g foo, bar, baz. It is not a list.

It splits the value on commas, and return stripped non-empty values:

```python
>>> os.environ['A'] = 'foo'
>>> os.environ['B'] = 'foo, bar, baz,'
>>> getter.getlist('a')
['foo']
>>> getter.getlist('b')
['foo', 'bar', 'baz']
```

def getbool(key[, default=False])
Retrieve a key from available configuration sources, and parse it as a boolean.
The following values are considered as True: 'on', 'yes', 'true', '1'. Case variations of those values also count as True.

def getint(key[, default=0])
Retrieve a key from available configuration sources, and parse it as an integer.

def getfloat(key[, default=0.0])
Retrieve a key from available configuration sources, and parse it as a floating point number.

def gettimedelta(key[, default='0d'])
Retrieve a key from available configuration sources, and parse it as a datetime.timedelta object.

def get_section(section_name)
Retrieve a dict-like proxy over a configuration section. This is intended to avoid polluting settings.py with a bunch of FOO = config.get('bar.foo'); BAR = config.get('bar.bar') commands.

Note: The returned object only supports the __getitem__ side of dicts (e.g. section_config['foo'] will work, 'foo' in section_config won't)


def get_ini_template()
Return INI like commented content equivalent to the default values.

For example:

```python
>>> getter.getlist('section.bar', default=['a', 'b'])
['a', 'b']
>>> getter.getbool('foo', default=True, doc="Set foo to True to enable the Truth")
```
True

```python
>>> print(g.get_ini_template())
[DEFAULT]
; NAMESPACE_FOO - type=bool - Set foo to True to enable the Truth
; foo = on

[section]
; NAMESPACE_SECTION_BAR - type=list
; bar = a, b
```

**Note:** This template is generated based on the `getxxx` calls performed on the ConfigGetter. If some calls are optional, the corresponding options might not be present in the `get_ini_template` return value.

### Example

With the following setup:

```python
# test_config.py
import getconf
cfg = getconf.ConfigGetter('getconf', ['/etc/getconf/example.ini'])

print("Env: %s" % cfg.getstr('env', 'dev'))
print("DB: %s" % cfg.getstr('db.host', 'localhost'))
print("Debug: %s" % cfg.getbool('dev.debug', False))
```

`# /etc/getconf/example.ini`

```ini
[DEFAULT]
env = example

[db]
host = foo.example.net
```

`# /etc/getconf/production.ini`

```ini
[DEFAULT]
env = prod

[db]
host = prod.example.net
```

We get the following outputs:

```bash
# Default setup
$ python test_config.py
Env: example
DB: foo.example.net
Debug: False

# Override 'env'
$ GETCONF_ENV=alt python test_config.py
Env: alt
DB: foo.example.net
Debug: False

# Override 'dev.debug'
$ GETCONF_DEV_DEBUG=on python test_config.py
```
Env: example
DB: foo.example.net
Debug: True

# Read from an alternate configuration file
$ GETCONF_CONFIG=/etc/getconf/production.ini python test_config.py
Env: prod
DB: prod.example.net
Debug: False

# Mix it up
$ GETCONF_DEV_DEBUG=on GETCONF_CONFIG=/etc/getconf/production python test_config.py
Env: prod
DB: prod.example.net
Debug: True

Advanced use

getconf supports some more complex setups; this document describes advanced options.

Recommended layout

Managing configuration can quickly turn into hell; here are a few guidelines:

- Choose where default values are stored
- Define how complex system-wide setup may get
- Decide whether local, development configuration is needed
- And whether user-local overrides are relevant

<table>
<thead>
<tr>
<th>Use case</th>
<th>Example program</th>
<th>Defaults storage</th>
<th>System-wide</th>
<th>Path-based</th>
<th>User-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-user binary</td>
<td>screen, bash</td>
<td>Within the code</td>
<td>Optional</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Folder-based soft</td>
<td>git, hg, ...</td>
<td>Within the code</td>
<td>Optional</td>
<td>Yes</td>
<td>Yes (global settings)</td>
</tr>
<tr>
<td>System daemon</td>
<td>uwsgi, ...</td>
<td>Default file with package</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Webapp</td>
<td>sentry, ...</td>
<td>Within the code</td>
<td>Yes</td>
<td>Yes (for dev)</td>
<td>No</td>
</tr>
</tbody>
</table>

This would lead to:

- End-user binary: `ConfigGetter('vim', ['/etc/vimrc', '~/.vimrc'])`
- Folder-based (git): `ConfigGetter('git', ['/etc/gitconfig', '~/.git/config', './.git/config'])`
- System daemon: `ConfigGetter('uwsgi', ['/usr/share/uwsgi/defaults.ini', '/etc/uwsgi/conf.d'])`
- Webapp: `ConfigGetter('sentry', ['/etc/sentry/conf.d/', '/dev_settings.ini'], defaults=sentry_defaults)`
**Defaults**

The default value may be provided in three different ways:

**Upon access** Use the `default` parameter of getters:

```
config.getstr('db.host', default='localhost')
```

This is pretty handy when all configuration values are read once and stored in another object. However, if the `ConfigGetter` object is the reference “configuration-holder” object, repeating the default at each call is a sure way to get mismatches between various code sections.

**Using a `defaults` directory** The constructor for `ConfigGetter` takes an extra keyword argument, `defaults`, that is used after all provided configuration files:

```
import getconf
config = getconf.ConfigGetter('myproj', ['~/.myproj.ini', '/etc/myproj.ini'], defaults={'logging': {'target': 'stderr'}})
```

With the above setup, `config.getstr('logging.target')` will be set to `stderr` if no value is provided through the environment nor the configuration files.

**In a package-owned configuration file** For complex projects, the list of settings can get huge. In those cases, it may be useful to provide a default configuration file alongside the package, with each option documented.

This default configuration file can also be used as a default, reference file:

```
import os
import getconf

# If we're in mymod/cli.py, config is at mymod/config/defaults.ini
filepath = os.path.abspath(__file__)
default_config = os.path.join(os.path.dirname(filepath), 'config', 'defaults.ini')
config = getconf.ConfigGetter('mymod', [default_config, '/etc/mymod.ini', '~/.mymod.ini'])
```

With the above setup, the package-provided `defaults.ini` will be used as defaults.

---

**Note:** Don’t forget to include the `defaults.ini` file in your package, with `setup.py`’s `include_package_data=True` and `MANIFEST.in`’s `include mymod/config/defaults.ini`.

---

**Configuration files in a folder**

For complex production projects, a common pattern is to split configuration among several files – for instance, a standard file holds logging settings, a platform-dependent one provides standard system paths, an infrastructure-related one has all server host/port pairs, and a secured one contains the passwords.

In order to support this pattern, `getconf`’s `config_files` list accepts folders as well; they are automatically detected on startup (using `os.path.isdir`).

With the following layout:

```
/etc
  -- myproj
     -- .keepdir
     -- 01_logging.ini
     -- 02_passwords.ini
```

Just setup your getter with `config = getconf.ConfigGetter('myproj', ['/etc/myproj/', ' ~/.config/myproj.ini'])`; this is strictly equivalent to using `config =
getconf.ConfigGetter('myproj', ['01_logging.ini', '02_passwords.ini', '~/.config/myproj.ini']).

Note: Remember: ConfigGetter parses configuration files in order this means that files provided at the beginning of the list are overridden by the next ones.

This aligns with the natural alphabetical handling of files: when using a folder, we want definitions from 99_overrides to override those in 00_base.

Precedency

When reading configuration from multiple sources, it can be complex to determine which source overrides which.

getconf’s precedence rules should be natural and easy to understand:

- Environment variables ALWAYS override other sources
- Configuration files are parsed in the order they are declared (last declaration wins)
- global defaults (in ConfigGetter(defaults=())) come before calling-defaults (in config.getstr(‘x.y’, default=‘blah’)), which come last.

Two special cases need to be handled:

- The environment-provided configuration file (<NAMESPACE>_CONFIG) has precedency over configuration files declared in ConfigGetter(config_files=[])
- When a configuration file is actually a directory (even if provided through <NAMESPACE>_CONFIG), its directly contained files are inserted in ALPHABETICAL ORDER, so that 99_foo actually overrides 10_base.

Example

Note: This example is an extremely complex layout, for illustration purposes. Understanding it might hurt your head. Please prefer simpler layouts!

With the following layout:

```
/etc
-- myproj.conf
-- myproj
  | -- .keepdir
  | -- 10_logging.ini
  | -- 20_passwords.ini
-- myproj.local
  -- .keepdir
  -- 15_logging.ini
  -- 20_passwords.ini
```

And the following environment variables:

```
MYPROJ_CONFIG=/etc/myproj.local
MYPROJ_DB_HOST=localhost
```

And this ConfigGetter setup:
import getconf

config = getconf.ConfigGetter('myproj', ['/etc/myproj.conf', '/etc/myproj'], defaults={'db': {'host': 'remote', 'port': '5432'}})

Then:

• config.getstr('db.host') is read from MYPROJ_DB_HOST=localhost
• config.getstr('db.name', 'foo') looks, in turn:
  – At /etc/myproj.local/20_passwords.ini's [db] name =
  – At /etc/myproj.local/15_logging.ini's [db] name =
  – At /etc/myproj/20_passwords.ini's [db] name =
  – At /etc/myproj/10_logging.ini's [db] name =
  – At /etc/myproj.conf's [db] name =
    – Defaults to foo
• config.getstr('db.port', '1234') looks, in turn:
  – At /etc/myproj.local/20_passwords.ini's [db] port =
  – At /etc/myproj.local/15_logging.ini's [db] port =
  – At /etc/myproj/20_passwords.ini's [db] port =
  – At /etc/myproj/10_logging.ini's [db] port =
  – At /etc/myproj.conf's [db] port =
    – Defaults to defaults['db']['port'] = '5432'

Goals

getconf aims to solve a specific problem: provide a simple way to load settings in a platform-typical manner.

The problem

Daemons and centralized applications need to fetch some platform-specific configuration to run:

• Mode of operation (debug vs. production vs. packaging)
• Address of remote services (databases, other servers, ...)
• Credentials

Beyond those required settings, an application needs to configure its behavior (timeouts, retries, languages, ...).

Various solutions exist:

• Command line flags
• Environment variables
• Files in /etc
The approach

getconf has been designed to provide the following features:

**Readability:**
- All options can be defined in a single file
- The provided values are typechecked (int, float, ...)
- All settings can have a default

**Development:**
- If I checkout the code and execute my program’s entry point, it should be able to start
- If my local setup is slightly different from the default (non-standard DB port, ...), I just have to put a simple `local_settings.ini` file in the current directory

**Continuous integration:** The continuous integration server just needs to set a few well-defined environment variables to point the program to the test databases, servers, ...

**Production:**
- In a could-like setup, I can use facilities provided by my platform to set the appropriate environment variables
- In a simpler, dedicated server setup, the application can also be configured with files in `/etc`

Other options

While designing getconf, we looked at other options:

**Define everything in files**
- This makes it difficult to override a single setting (where should the file be?)
- Not compatible with env-based cloud platforms
- dev and prod often have very different configurations, but flat files don’t provide a simple switch to set those defaults

**Define everything in the environment** Requires a prod-like setup for starting local servers, with files listing the environment variables

**Load a single file, which includes others**
- Quickly turns into a maze of “local includes dev includes base”
- Hard to see where a setting is defined

Development

Clone the repository and install the development dependencies in a virtualenv:

```
pip install -r requirements_dev.txt
```

To run tests:

```
nosetests
```
ChangeLog

Next version

1.7.0 (2017-02-23)

New:
- Allow using an empty namespace (ConfigGetter(namespace=getconf.NO_NAMESPACE) to load un-prefixed environment variables.

1.6.0 (2017-02-03)

New:
- Remove support for string interpolation in .ini file If this undocumented getconf feature is still needed by some users, we might consider restoring it in a future release.

1.5.2 (2017-01-23)

New:
- Add a new gettimedelta function to parse simple durations expressed as strings (10 days as ‘10d’, 3 hours as ‘3h’, etc.)

1.5.1 (2016-12-15)

New:
- Display the key of the value that triggers an error to help resolve.

1.5.0 (2016-05-11)

New:
- Better AssertionError messages when default values have the wrong type.
- Add ConfigGetter.get_ini_template() method

1.4.1 (2015-08-28)

New:
- Improve error reporting when raising on wrongly typed defaults

1.4.0 (2015-08-27)

New:
- Enforce type checking on every getconf.getXXX() call
- Add getconf.getstr() method
• Enable using None as default value for every function
• Better support for Python 3.3, 3.4 and wheel distribution

Deprecated:
• Use of strings as default values for getconf.getlist()
• Use of getconf.get() in favor of getconf.getstr()

1.3.0 (2015-04-14)

New:
• Add getfloat() method
• Allow globs in config_files
• <PROJECT>_CONFIG env var will now have the same behaviour than config_files items

1.2.1 (2014-10-24)

Bugfix:
• Fix version number

1.2.0 (2014-10-20)

New:
• Add support for directory-based configuration and providing defaults through a dict

Removed:
• Remove support for ConfigGetter(namespace, file1, file2, file3) syntax (deprecated in 1.1.0), use ConfigGetter(namespace, [file1, file2, file3]) instead

1.1.0 (2014-08-18)

New:
• New initialization syntax

Deprecated
• Using argument list for config file paths when initializing ConfigGetter is now deprecated, you need to use a list (use ConfigGetter(namespace, ['settings_1.ini', 'settings_2.ini']) instead of ConfigGetter(namespace, 'settings_1.ini', 'settings_2.ini'))

1.0.1 (2014-04-13)

Bugfix:
• Fix packaging (missing requirements files)
1.0.0 (2014-04-12)

New:

- First version
Indices and tables

- genindex
- modindex
- search
g
getconf, 13
Index

C
ConfigGetter (class in getconf), 13

G
get_ini_template() (getconf.ConfigGetter method), 14
cgetsection() (getconf.ConfigGetter method), 14
ggetbool() (getconf.ConfigGetter method), 14
ggetconf (module), 13
ggetfloat() (getconf.ConfigGetter method), 14
ggetint() (getconf.ConfigGetter method), 14
ggetlist() (getconf.ConfigGetter method), 14
ggetstr() (getconf.ConfigGetter method), 13
ggettimedelta() (getconf.ConfigGetter method), 14