pyfarm.core Documentation

Release 0.8.2

Oliver Palmer, Guido Winkelmann

June 22, 2014

This package contains some shared libraries and objects which other parts of PyFarm, such as `pyfarm.master` and `pyfarm.agent`, use directly.

Note: While this code could be used directly, it's primarily intended to be inside of other parts of PyFarm.

Contents

CHAPTER 1

pyfarm.core package

1.1 PyFarm Core

Core library used by other components of PyFarm.

1.2 Submodules

1.2.1 pyfarm.core.config module

Configuration Object

Basic module used for reading configuration data into PyFarm in various forms.

const BOOLEAN_TRUE set of values which will return a True boolean value from `read_env_bool()`

const BOOLEAN_FALSE set of values which will return a False boolean value from `read_env_bool()`

class `pyfarm.core.config.`**`Configuration`**(*name*, *version=None*)

Bases: `builtins.dict`

Main object responsible for finding, loading, and merging configuration data. By default this class does nothing until `load()` is called. Once this method is called `Configuration` class will populate itself with data loaded from the configuration files. The configuration files themselves can be loaded from multiple location depending on the system's setup. For example on Linux you might end up attempting to load:

- The default configuration as provided by PyFarm's source.
- `/etc/pyfarm/agent/agent.yml`
- `/etc/pyfarm/agent/1/agent.yml`
- `/etc/pyfarm/agent/1.2/agent.yml`
- `/etc/pyfarm/agent/1.2.3/agent.yml`
- `~/.pyfarm/agent/agent.yml`
- `~/.pyfarm/agent/1/agent.yml`
- `~/.pyfarm/agent/1.2/agent.yml`
- `~/.pyfarm/agent/1.2.3/agent.yml`

- `etc/pyfarm/agent/agent.yml`
- `etc/pyfarm/agent/1/agent.yml`
- `etc/pyfarm/agent/1.2/agent.yml`
- `etc/pyfarm/agent/1.2.3/agent.yml`

`Configuration` will only attempt to load data from files which exist on the file system when `load()` is called. If multiple files exist the data will be loaded from each file with the successive data overwriting the value from the previously loaded configuration file. So if you have two files containing the same data:

- `/etc/pyfarm/agent/agent.yml`

  ::

  env: a: 0

  foo: 1 bar: true

- `etc/pyfarm/agent/1.2.3/agent.yml`

  ::

  env: a: 1 b: 1

  foo: 0

You'll end up with a single merged configuration. Please note that the only keys which will be merged in the configuration are the `env` key. Configuration files are meant to store simple data and while it can be used to store more complicate data it won't merge any other data structures.

::

env: a: 1 b: 1

foo: 0 bar: true

Variables

- **DEFAULT_SYSTEM_ROOT** (*string*) – The system level directory that we should look for configuration files in. This path is platform dependent:
  - **Linux** - /etc/
  - **Mac** - /Library/
  - **Windows** - %ProgramData%. An environment variable that varies depending on the Windows version. See Microsoft's docs: https://www.microsoft.com/security/portal/mmpc/shared/variables.aspx

  The value built here will be copied onto the instance as `system_root`
- **DEFAULT_USER_ROOT** (*string*) – The user level directory that we should look for configuration files in. This path is platform dependent:
  - **Linux/Mac** - ~ (home directory)
  - **Windows** - %APPDATA%. An environment variable that varies depending on the Windows version. See Microsoft's docs: https://www.microsoft.com/security/portal/mmpc/shared/variables.aspx

  The value built here will be copied onto the instance as `user_root`
- **DEFAULT_FILE_EXTENSION** (*string*) – The default file extension of the configuration files. This will default to `.yml` and will be copied to `file_extension` when the class is instanced.

- **DEFAULT_LOCAL_DIRECTORY_NAME** (*string*) – A directory local to the current process which we should search for configuration files in. This will default to `etc` and will be copied to `local_dir` when the class is instanced.
- **DEFAULT_PARENT_APPLICATION_NAME** (*string*) – The base name of the parent application. This used used to build child directories and will default to `pyfarm`.
- **DEFAULT_ENVIRONMENT_PATH_VARIABLE** (*string*) – A environment variable to search for a configuration path in.

Parameters

- **name** (*string*) – The name of the configuration itself, typically 'master' or 'agent'. This may also be the name of a package such as 'pyfarm.agent'. When the package name is provided we can usually automatically determine the version number.
- **version** (*string*) – The version the version of the program running.

`DEFAULT_ENVIRONMENT_PATH_VARIABLE` = **'PYFARM_CONFIG_ROOT'**

`DEFAULT_FILE_EXTENSION` = **'.yml'**

`DEFAULT_LOCAL_DIRECTORY_NAME` = **'etc'**

`DEFAULT_PARENT_APPLICATION_NAME` = **'pyfarm'**

`DEFAULT_SYSTEM_ROOT` = **'/etc'**

`DEFAULT_USER_ROOT` = **'/home/docs'**

`directories`()
Returns a list of platform dependent directories which may contain configuration files.

`files`()
Returns a list of configuration files.

`load`(*environment=None*)
Loads data from the configuration files. Any data present in the `env` key in the configuration files will update **:arg:'environment'**.

Parameters **environment** (*dict*) – A dictionary to load data in the `env` key from the configuration files into. This would typically be set to **:var:'os.environ'** so the environment itself could be updated.

`split_version`(*sep='.'*)
Splits `self.version` into a tuple of individual versions. For example `1.2.3` would be split into `['1', '1.2', '1.2.3']`

`pyfarm.core.config.`**`read_env`**(*envvar*, *default=<object object at 0x7f8df267c070>*, *warn_if_unset=False*, *eval_literal=False*, *raise_eval_exception=True*, *log_result=True*, *desc=None*)
Lookup and evaluate an environment variable.

Parameters

- **envvar** (*string*) – The environment variable to lookup in `os.environ`
- **default** (*object*) – Alternate value to return if `envvar` is not present. If this is instead set to `NOTSET` then an exception will be raised if `envvar` is not found.
- **warn_if_unset** (*bool*) – If True, log a warning if the value being returned is the same as `default`
- **eval_literal** – if True, run `literal_eval()` on the value retrieved from the environment

- **raise_eval_exception** (*bool*) – If True and we failed to parse `envvar` with `literal_eval()` then raise a `EnvironmentKeyError`
- **log_result** (*bool*) – If True, log the query and result to INFO. If False, only log the query itself to DEBUG. This keyword mainly exists so environment variables such as `PYFARM_SECRET` or `PYFARM_DATABASE_URI` stay out of log files.
- **desc** (*string*) – Describes the purpose of the value being returned. This may also be read in at the time the documentation is built.

`pyfarm.core.config.`**`read_env_bool`**`(`**args*, ***kwargs*`)`

Wrapper around `read_env()` which converts environment variables to boolean values. Please see the documentation for `read_env()` for additional information on exceptions and input arguments.

Raises

- **AssertionError** – raised if a default value is not provided
- **TypeError** – raised if the environment variable found was a string and could not be converted to a boolean.

`pyfarm.core.config.`**`read_env_number`**`(`**args*, ***kwargs*`)`

Wrapper around `read_env()` which will read a numerical value from an environment variable. Please see the documentation for `read_env()` for additional information on exceptions and input arguments.

Raises TypeError raised if we either failed to convert the value from the environment variable or the value was not a float, integer, or long

`pyfarm.core.config.`**`read_env_strict_number`**`(`**args*, ***kwargs*`)`

Strict version of `read_env_number()` which will only return an integer

Parameters number_type – the type of number(s) this function must return

Raises

- **AsssertionError** – raised if the number_type keyword is not provided (required to check the type on output)
- **TypeError** – raised if the type of the result is not an instance of *number_type*

1.2.2 pyfarm.core.enums module

Enums

Provides enum values for certain aspect of PyFarm. See below for more detailed information.

Operating System

Describes an operating system type.

Table 1.1: OperatingSystem

Attribute	Description
LINUX	operating system on agent is a Linux variant
WINDOWS	operating system on agent is a Windows variant
MAC	operating system on agent is an Apple OS variant

Agent State

The last known state of the remote agent, used for making queue decisions and locking off resources.

Attribute	Description
OFFLINE	agent cannot be reached
ONLINE	agent is waiting for work
DISABLED	agent is online but cannot accept work
RUNNING	agent is currently processing work
ALLOC	special internal state used when the agent entry is being built

Work State

The state a job or task is currently in. These values apply more directly to tasks as job statuses are built from task status values.

Attribute	Description
PAUSED	this task cannot be assigned right now but can be once unpaused
RUNNING	work is currently being processed
DONE	work is finished (previous failures may be present)
FAILED	work as failed and cannot be continued

Use Agent Address

Describes which address should be used to contact the agent

Attribute	Description
LOCAL	use the address which was provided by the agent
REMOTE	use the address which we received the request from
HOSTNAME	disregard both the local IP and the remote IP and use the hostname
PASSIVE	agent cannot be contacted but will still request work and process jobs

const PY_MAJOR the major Python version

const PY_MINOR the minor Python version

const PY_VERSION a tuple containing the major and minor Python versions

const PY3 True if running Python 3

const PY2 True if running Python 2

const PY26 True if running Python 2.6

const PY27 True if running Python 2.7

const NOTSET Instance of the object class, mainly used when None is actually a valid value

const STRING_TYPES A tuple of string types, provided for Python 3 backwards compatibility

const NUMERIC_TYPES A tuple of numeric types, provided for Python 3 backwards compatibility

const INTEGER_TYPES A tuple of integer types, provided for Python 3 backwards compatibility

const BOOLEAN_TRUE A set containing strings and other objects representing `True` under some conditions. Generally used by `pyfarm.core.utility.convert.bool()`

const BOOLEAN_FALSE A set containing strings and other objects representing `False` under some conditions. Generally used by `pyfarm.core.utility.convert.bool()`

const NONE A set containing strings and other objects which represent `None` under some conditions. Generally used by `pyfarm.core.utility.convert.none()`

const INTERACTIVE_INTERPRETER True when we're running inside an interactive interpreter such as a Python shell like IPython. This value will also be True if there's an active debugger.

const OS The current os type, the value will map to one of the values in `OperatingSystem`

const POSIX True if `OS in (OperatingSystem.LINUX, OperatingSystem.MAC)`

const WINDOWS True if `OS == OperatingSystem.WINDOWS`

const LINUX True if `OS == OperatingSystem.LINUX`

const MAC True if `OS == OperatingSystem.MAC`

`pyfarm.core.enums.`**`Enum`**`(`*classname*, ***kwargs*`)`

Produce an enum object using `namedtuple()`

```
>>> Foo = Enum("Foo", A=1, B=2)
>>> assert Foo.A == 1 and Foo.B == 2
>>> FooTemplate = Enum("Foo", A=int, instance=False)
>>> Foo = FooTemplate(A=1)
>>> assert Foo.A == 1
```

Parameters

- **classname** (*str*) – the name of the class to produce
- **to_dict** – a callable function to add to the named tuple for converting the internal values into a dictionary
- **instance** (*bool*) – by default calling `Enum()` will produce an instanced `namedtuple()` object, setting `instance` to False will instead produce the named tuple without instancing it

class `pyfarm.core.enums.`**`Values`**`(`**args*, ***kwargs*`)`

Bases: `pyfarm.core.enums.Values`

Stores values to be used in an enum. Each time this class is instanced it will ensure that the input values are of the correct type and unique.

`NUMERIC_TYPES` **= (<class 'int'>,)**

`check_uniqueness` **= True**

`pyfarm.core.enums.`**`cast_enum`**`(`*enum*, *enum_type*`)`

Pulls the requested `enum_type` from `enum` and produce a new named tuple which contains only the requested data

```
>>> from pyfarm.core.enums import Enum, Values
>>> FooBase = Enum("Foo", A=Values(int=1, str="1")
>>> Foo = cast_enum(FooBase, str)
>>> assert Foo.A == "1"
>>> Foo = cast_enum(FooBase, int)
>>> assert Foo.A == 1
>>> assert Foo._map == {"A": 1, 1: "A"}
```

Warning: This function does not perform any kind of caching. For the most efficient usage it should only be called once per process or module for a given enum and enum_type combination.

`pyfarm.core.enums.`**`operating_system`**`(`*plat='linux'*`)`

Returns the operating system for the given platform. Please note that while you can call this function directly you're more likely better off using values in `pyfarm.core.enums` instead.

1.2.3 pyfarm.core.files module

File Operations

Low level module responsible for working with and resolving information related to file paths

`pyfarm.core.files.`**`expandenv`**`(`*envvar*, *validate=True*, *pathsep=None*`)`

Takes the environment variable given by *envvar*, splits it into multiple paths, then expands/validates them as requested.

Parameters

- **validate** (*bool*) – if True then only paths which exist will be returned
- **pathsep** (*str*) – if provided then use this as the value to split the environment variable by, otherwise use *os.pathsep*

Raises

- **exceptions.EnvironmentError** – raised if the requested *envvar* does not exist
- **exceptions.ValueError** – raised if the requested *envvar* does exist but does not contain any data

`pyfarm.core.files.`**`expandpath`**`(`*path*`)`

Expands environment variables and user paths such as ~ in *path* using `os.path.expandvars()` and `os.path.expanduser()`

`pyfarm.core.files.`**`which`**`(`*program*`)`

returns the path to the requested program

Note: This function will not resolve aliases

Raises OSError raised if the path to the program could not be found

1.2.4 pyfarm.core.testutil module

1.2.5 pyfarm.core.utility module

Utilities

General utility functions that are not specific to individual components of PyFarm.

class `pyfarm.core.utility.`**`ImmutableDict`**`(`*iterable=None*, ***kwargs*`)`

Bases: `builtins.dict`

A basic immutable dictionary that's built on top of Python's standard `dict` class. Once `__init__()` has been run the contents of the instance can no longer be modified

`clear``(`**args*, ***kwargs*`)`

`pop``(`**args*, ***kwargs*`)`

`popitem``(`**args*, ***kwargs*`)`

setdefault(**args*, ***kwargs*)

update(**args*, ***kwargs*)

class `pyfarm.core.utility.`**PyFarmJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

encode(*o*)

class `pyfarm.core.utility.`**convert**

Bases: `builtins.object`

Namespace containing various static methods for converting data.

Some staticmethods are named the same as builtin types. The name indicates the expected result but the staticmethod may not behave the same as the equivalently named Python object. Read the documentation for each staticmethod to learn the differences, expected input and output.

static bool(*value*)

Converts `value` into a boolean object. This function mainly exits so human-readable booleans such as 'yes' or 'y' can be handled in a single location. Internally it does *not* use `bool()` and instead checks `value` against `pyfarm.core.enums.BOOLEAN_TRUE` and `pyfarm.core.enums.BOOLEAN_FALSE`.

Parameters value – The value to attempt to convert to a boolean. If this value is a string it will be run through `.lower().strip()` first.

Raises ValueError Raised if we can't convert `value` to a true boolean object

static bytetomb(*value*)

Convert bytes to megabytes

```
>>> convert.bytetomb(10485760)
10.0
```

static list(*value*, *sep='*, *'*, *strip=True*, *filter_empty=True*)

Converts `value` into a list object by splitting on `sep`.

Parameters

- **value** (*str*) – The string we should convert into a list
- **sep** (*str*) – The string that we should split `value` by.
- **strip** (*bool*) – If `True`, strip extra whitespace from the results so `'foo, bar'` becomes `['foo', 'bar']`
- **filter_empty** (*bool*) – If `True`, any result that evaluates to `False` will be removed so `'foo,,'` would become `['foo']`

static mbtogb(*value*)

Convert megabytes to gigabytes

```
>>> convert.mbtogb(2048)
2.0
```

static none(*value*)

Converts `value` into `None`. This function mainly exits so human-readable values such as 'None' or 'null' can be handled in a single location. Internally this checks `value` against `pyfarm.core.enums.NONE`

Parameters **value** – The value to attempt to convert to `None`. If this value is a string it will be run through `.lower().strip()` first.

Raises **ValueError** Raised if we can't convert `value` to `None`

static `ston`(*value*, *types=(<class 'int'>, <class 'float'>, <class 'complex'>)*)

Converts a string to an integer or fails with a useful error message

Parameters **value** (*string*) – The value to convert to an integer

Raises

- **ValueError** – Raised if `value` could not be converted using `literval_eval()`
- **TypeError** – Raised if `value` was not converted to a float, integer, or long

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index