qiutil

Release

Jul 24, 2017

Contents

CHAPTER 1

Synopsis

qiutil provides general-purpose utilities for the OHSU QIN projects.

API http://qiutil.readthedocs.org/en/latest/api/index.html

Git https://www.github.com/ohsu-qin/qiutil

CHAPTER 2

Feature List

1. Configuration file parser.
2. Logging configuration.
3. Command logging options.
4. Collection data structures and utilities.
5. File helper functions.
6. Simple UID generator.

CHAPTER 3

Installation

Add `qiutil` to your Python project setup.py `install_requires`.

CHAPTER 4

Development

Testing is performed with the nose package, which must be installed separately.

Documentation is built automatically by ReadTheDocs when the project is pushed to GitHub. Documentation can be generated locally as follows:

- Install Sphinx, if necessary.
- Run the following in the `doc` subdirectory:

```
make html
```

API Documentation

qiutil

qiutil.**__version__** = '2.3.1'
The *major.minor.patch* version. The verson numbering scheme is described in Fast and Loose Versioning

ast_config

class qiutil.ast_config.**ASTConfig**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*)
Bases: ConfigParser.ConfigParser

ASTConfig parses a configuration file with AST property values as follows:

•An unquoted digits value is parsed as an integer.

•An unquoted digits value with a single period is parsed as a float.

•A [] bracketed value is parsed as a list.

- A `()` bracketed value is parsed as a tuple.
- A `{}` bracketed value is parsed as a dictionary.
- A case-insensitive match on `true` or `false` is parsed as the Python object `True`, resp. `False`.
- A case-insensitive match on `none`, `null` or `nil` is parsed as the Python `None` object.
- All other values are parsed as string.

For example, given the configuration file `tuning.cfg` with content:

```
[Tuning]
method = FFT
iterations = [[1, 2], 5]
parameters = [(1,), (2, 3), -0.4]
two_tailed = false
threshold = 4.0
plugin_args = {'qsub_args': '-pe mpi 48-120'}
```

then:

```
>>> cfg = ASTConfig('tuning.cfg')
>>> cfg['Tuning']
{'method': u'FFT', 'parameters' = [(1,), (2, 3), -0.4],
 'iterations': [[1, 2], 5],
 'two_tailed': False, 'threshold': 4.0,
 'plugin_args': {'qsub_args': '-pe mpi 48-120'}}
```

COLL_PAT = '\n \\%(left)s # The left delimiter\n (.*) # The list items\n \\%(right)s$ # The right delimiter\n '
A bunch string pattern.

DICT_PAT = <_sre.SRE_Pattern object>
A dictionary string pattern.

EMBEDDED_DICT_PAT = <_sre.SRE_Pattern object>
A (prefix)(dictionary)(suffix) recognition pattern.

EMBEDDED_LIST_PAT = <_sre.SRE_Pattern object>
A (prefix)(list)(suffix) recognition pattern.

EMBEDDED_TUPLE_PAT = <_sre.SRE_Pattern object>
A (prefix)(tuple)(suffix) recognition pattern.

LIST_PAT = <_sre.SRE_Pattern object>
A list string pattern.

PARSEABLE_ITEM_PAT = <_sre.SRE_Pattern object at 0x137a8d0>
A non-list string parseable by AST.

TUPLE_PAT = <_sre.SRE_Pattern object>
A tuple string pattern.

exception `qiutil.ast_config.`**`ConfigError`**
Bases: `exceptions.Exception`

Configuration parsing error.

`qiutil.ast_config.`**`read_config`**(**locations*)
Reads and parses the given configuration files.

Parameters **`locations`** – the input configuration file paths

Returns the configuration

Return type *qiutil.ast_config.ASTConfig*

Raises **IOError** – if no configuration files could not be read

collections

qiutil.collections.**EMPTY_DICT = {}**
An immutable empty dictionary. This constant serves as an efficient method return default value.

class qiutil.collections.**ImmutableDict**(**args*, ***kwargs*)
Bases: dict

ImmutableDict is a dictionary that cannot be changed after creation.

An ImmutableDict is *not* hashable and therefore cannot be used as a dictionary key or set member. See http://www.python.org/dev/peps/pep-0351 for the rationale.

__init__(**args*, ***kwargs*)

qiutil.collections.**concat**(**iterables*)

Parameters **iterables** – the iterables to concatenate

Returns the concatenated list

Return type list

qiutil.collections.**is_nonstring_iterable**(*value*)

Parameters **value** – the object to check

Returns whether the given value is a non-string iterable object

qiutil.collections.**nested_defaultdict**(*factory*, *levels=0*)
Makes a defaultdict for the given factory and number of levels, e.g.:

```
>> from qiutil.collections import nested_defaultdict as dd
>> dd(list, 0)[1]
[]
>> dd(dict, 2)[1][2][3]
{}
```

Note that the default levels parameter value 0 is synonymous with the standard Python collections defaultdict, i.e.:

```
dd(list)
```

is the same as:

```
dd(list, 0)
```

or:

```
from collections import defaultdict
defaultdict(list)
```

Thus, this nested_defaultdict function can serve as a drop-in replacement for defaultdict.

Parameters

- **factory** – the 0th level defaultdict factory.
- **levels** – the number of levels

qiutil.collections.**to_series**(*items*, *conjunction='and'*)

Formats the given items as a series string.

Example:

```
>>> to_series([1, 2, 3])
'1, 2 and 3'
```

Parameters

- **items** – the items to format in a series
- **conjunction** – the series conjunction

Returns the items series

Return type str

qiutil.collections.**tuplize**(*iterable*)

Recursively creates nested tuples from the given iterable object.

Parameters **iterable** – the iterable to convert

Returns the comparable tuple

qiutil.collections.**update**(*target*, **sources*, ***opts*)

Updates the given target object from the given source objects. The target object can be a dictionary, list or set. The target and sources are validated for compatibility as follows:

•If the target object is a Mapping, then the sources must be Mappings.

•Otherwise, if the target object is a list or set, then the sources must be non-string iterables.

The target is updated from the sources in order as follows:

•If the target object is a Mapping and the *recursive* flag is falsey, then the standard Python dictionary update is applied.

•If the target object is a Mapping and the *recursive* flag is truthy, then the update is applied recursively to nested dictionaries, e.g.:

>> from qiutil.collections import update >> target = dict(a=dict(aa=1)) >> update(target, dict(a=dict(aa=2, ab=3))) >> target {'a': {'aa': 2, 'ab': 3}}

•If the target object is a list or set, then the source items which are not yet in the target are added to the target, e.g.:

>> from qiutil.collections import update >> target = [1, 2, 2, 5] >> update(target, [4, 2, 6, 6]) >> target [1, 2, 2, 5, 4, 6, 6]

This function adapts the solution offered in a *StackOverflow post <http://stackoverflow.com/questions/3232943/update-value-of-a-nested-dictionary-of-varying-depth>* to support lists, sets and multiple sources.

Parameters

- **target** – the dictionary to update
- **sources** – the update source dictionaries
- **opts** – the following keyword options:
- **recursive** – if True, then apply the update recursively to nested dictionaries

command

Command helper functions.

qiutil.command.**add_options**(*parser*)
Adds the standard `--log`, `--quiet`, `--verbose` and `--debug` options to the given command line argument parser.

qiutil.command.**configure_log**(**names*, ***opts*)
Configures the logger.

Parameters

- **names** – the loggers to configure
- **opts** – the following keyword options:
- **log** – the log file
- **log_level** – the log level

dictionary_hierarchy

class qiutil.dictionary_hierarchy.**DictionaryHierarchy**(*root*)
Bases: `object`

A DictionaryHierarchy wraps a nested dictionary. The DictionaryHierarchy iterator enumerates the root -> leaf paths.

For example, the hierarchy of a nested dictionary given by:

```
1 : 'a'
2 :
  3 : 'b'
  4 : 'c', 'd'
  6 :
    7 : 'e'
8 : 'f'
```

results in the following paths:

```
1, 'a'
2, 3, 'b'
2, 4, 'c'
2, 4, 'd'
2, 6, 7, 'e'
8, 'f'
```

Parameters **root** (*dict*) – the nested dictionary to wrap by this hierarchy

Raises **ArgumentError** – if the given root is not a dictionary

__init__(*root*)

Parameters **root** (*dict*) – the nested dictionary to wrap by this hierarchy

Raises **ArgumentError** – if the given root is not a dictionary

qiutil.dictionary_hierarchy.**on**(*root*)

Parameters **root** – the nested dictionary to wrap

Returns a new DictionaryHierarchy on the given root dictionary

file

class qiutil.file.**FileIterator**(**filespecs*)
Bases: object

FileIterator is a generator class which iterates over the files contained recursively in the initializer *filespecs* parameters.

Parameters **filespecs** – the files, directories or file generators over which to iterate

__init__(**filespecs*)

Parameters **filespecs** – the files, directories or file generators over which to iterate

class qiutil.file.**Finder**(*glob='*'*, *regex='.*'*)
Bases: object

Finder matches a file name glob pattern and regular expression.

Parameters

- **glob** – the glob pattern string
- **regex** – the RegExp object or pattern string

__init__(*glob='*'*, *regex='.*'*)

Parameters

- **glob** – the glob pattern string
- **regex** – the RegExp object or pattern string

find(*base_dir=None*)
Iterates over the files which match both the *glob* and the *regex*. Each iteration result is an absolute file path.

Param the parent base directory path (default current directory)

Yield the next matching absolute file path

glob = None
The glob pattern string (default *).

match(*base_dir=None*)
Iterates over the files which match both the *glob* and the *regex*. The match result does not include the base directory.

Param the parent base directory path (default current directory)

Yield the next match

regex = None
The file match regular expression (default .*).

qiutil.file.**SPLITEXT_PAT = <_sre.SRE_Pattern object>**
Regexp pattern that splits the name and extension.

See *splitexts()*

qiutil.file.**generate_file_name**(*ext=None*)
Makes a valid file name which is unique to within one microsecond of calling this function. The file name is ten characters long without the extension.

Parameters **ext** – the optional file extension, with leading period delimiter

Returns the file name

qiutil.file.**open**(**args*, ***kwds*)
Opens the given file. If the file extension ends in `.gz`, then the content is uncompressed.

Parameters **filename** – the file path

Returns the file input stream

Raise IOError if the file cannot be read

qiutil.file.**splitboth**(*location*)
Splits the given file path into a directory, name and extension. Unlike `os.path.splitext`, the resulting extension can be composite.

Example:

```
>>> import os
>>> from qiutil.file import splitall
>>> splitall('/tmp/foo.nii.gz')
('/tmp', 'foo', '.nii.gz')
```

Parameters **location** – the file path

Returns the *(directory, prefix, extensions)* tuple

qiutil.file.**splitexts**(*location*)
Splits the given file path into a name and extension. Unlike `os.path.splitext`, the resulting extension can be composite.

Example:

```
>>> import os
>>> os.path.splitext('/tmp/foo.nii.gz')
('/tmp/foo.nii', '.gz')
>>> from qiutil.file import splitexts
>>> splitexts('/tmp/foo.3/bar.nii.gz')
('/tmp/foo.3/bar', '.nii.gz')
```

Parameters **location** – the file path

Returns the *(prefix, extensions)* tuple

logging

qiutil.logging.**BASE_DIR = '/home/docs/checkouts/readthedocs.org/user_builds/qiutil/envs/stable/local/lib/python2.7/site**
The `qiutil` package directory.

qiutil.logging.**DEF_LOG_CFG = '/home/docs/checkouts/readthedocs.org/user_builds/qiutil/envs/stable/local/lib/python2.7**
The default logging configuration file path.

qiutil.logging.**LOG_CFG_ENV_VAR = 'LOG_CONFIG'**
The user-defined environment variable logging configuration file path.

qiutil.logging.**LOG_CFG_FILE = 'logging.yaml'**
The optional current working directory logging configuration file name.

qiutil.logging.**configure**(**names*, ***opts*)

Configures logging. The logging configuration is obtained from from the given keyword arguments and the YAML logging configuration files. The following logging configuration files are loaded in low-to-high precedence:

- the `qiutil` module `conf/logging.yaml` file
- the `logging.yaml` file in the current directory
- the file specified by the `LOG_CFG` environment variable
- the *cfg_file* parameter

The `opts` keyword arguments specify simple logging parameters that override the configuration file settings. The keyword arguments can include the *filename* and *level* short-cuts, which are handled as follows:

- if the *filename* is None, then file logging is disabled. Otherwise, the file handler file name is set to the *filename* value.
- The *level* is set for the logger. In addition, if the logger has a file handler, then that file handler level is set. Otherwise, the console handler level is set.

The logging configuration file `formatters`, `handlers` and `loggers` sections are updated incrementally. For example, the `conf/logging.yaml` source distribution file defines the `default` formatter `format` and `datefmt`. If the `logging.yaml` file in the current directory overrides the `format` but not the `datefmt`, then the default `datefmt` is retained rather than unset. Thus, a custom logging configuration file need define only the settings which override the default configuration.

By default, `ERROR` level messages are written to the console. If the log file is set, then the default logger writes `INFO` level messages to a rotating log file.

If the file handler is enabled, then this *qiutil.logging.configure()* method ensures that the log file parent directory exists.

Examples:

- Write to the log:

```
>>> from qiutil.logging import logger
>>> logger(__name__).debug("Started the application...")
```

or, in a class instance:

```
>>> from qiutil.logging import logger
>>> class MyApp(object):
...     def __init__(self):
...         self._logger = logger(__name__)
...     def start(self):
...         self._logger.debug("Started the application...")
```

- Write debug messages to the file log:

```
>>> import qiutil
>>> qiutil.logging.configure(level='DEBUG')
```

- Set the log file:

```
>>> import qiutil
>>> qiutil.logging.configure(filename='log/myapp.log')
```

- Define your own logging configuration:

```
>>> import qiutil
>>> qiutil.logging.configure('/path/to/my/conf/logging.yaml')
```

•Simplify the console log message format by creating the following `./logging.yaml` customization:

```
---
formatters:
  simple:
    format: '%(name)s - %(message)s'
handlers:
  console:
    formatter: simple
```

Parameters

- **names** – the logging contexts (default root)
- **opts** – the Python `logging.conf` options, as well as the following short-cuts:
- **config** – the custom configuration YAML file
- **filename** – the log file path
- **level** – the file handler log level

`qiutil.logging.`**`logger`**`(`*name*`)`

This method is the preferred way to obtain a logger.

Example:

```
>>> from qiutil.logging import logger
>>> logger(__name__).debug("Starting my application...")
```

Note Python `nosetests` captures log messages and only reports them on failure.

Parameters **name** – the caller's context `__name__`

Returns the Python Logger instance

uid

`qiutil.uid.`**`MODULO_FACTOR = {'h': 0.016666666666666666, 's': 1, 'u': 1000000, 'm': 1000, 'd': 0.0006944444444444445}`**

The *generate_uid()* modulo unit factors.

`qiutil.uid.`**`generate_string_uid`**`(**`*opts*`)`

Makes a string id based on the *generate_uid()* value. The string id is a URL-safe base64-encoded string without trailing filler or linebreak. It is thus suitable for file names as well as URLs.

The generated id is ten characters long for the default base year.

Parameters **opts** – the *generate_uid()* options

Return type str

Returns the string UID

`qiutil.uid.`**`generate_uid`**`(**`*opts*`)`

Makes an id that is unique modulo a given duration unit. The id is unique to within one duration unit of calling this function for the execution context which runs this method. The default modulo unit is microseconds (`u`). Other supported durations are milliseconds (`m`), seconds (`s`), hours (`h`) and days (`d`).

This is a simple, fast id generator that is adequate for most purposes.

The base year parameter should be no greater than the current year and should not change over the lifetime of all objects created using that year. A higher base year results in a smaller UID.

Parameters

- **opts** – the following options:
- **modulo** – the modulo unit (default 'u')

Option modulo the optional modulo unit parameter

Option year the optional base year for the time offset (default 2001)

Return type long

Returns the UID

which

qiutil.which.**which**(*program*)

Returns the file path of the first executable of the given program name in the system PATH environment variable.

This function is a system-independent Python equivalent of the *nix `which` command.

Note The implementation is adapted from http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python.

Parameters **program** – the program name to check

Returns the filename in the system path

Python Module Index

q

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

W