Skeletor Documentation

Release 0.1.1

David Aguilar

January 30, 2015

Contents

skeletor is a Python library for reusable SQLAlchemy utilities. It contains modules that can be considered a minimalist "application skeleton", thus "skeletor".

Contents

CHAPTER 1

Usage

CHAPTER 2

Download & Install

The easiest way to get skeletor is via PyPi with pip:

```
$ pip install -U skeletor
```

You can also download or *clone* the latest code and install from source:

```
$ python setup.py install
```

CHAPTER 3

API Reference

3.1 skeletor API documenation

Contents

3.1.1 skeletor.core – Skeleton core components

skeletor.core.compat – Compatibility adapters

skeletor.core.compat.**bytes**
alias of str

class skeletor.core.compat.**set**
set() -> new empty set object set(iterable) -> new set object

Build an unordered collection of unique elements.

add()
Add an element to a set.

This has no effect if the element is already present.

clear()
Remove all elements from this set.

copy()
Return a shallow copy of a set.

difference()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

difference_update()
Remove all elements of another set from this set.

discard()
Remove an element from a set if it is a member.

If the element is not a member, do nothing.

intersection()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

intersection_update()
Update a set with the intersection of itself and another.

isdisjoint()
Return True if two sets have a null intersection.

issubset()
Report whether another set contains this set.

issuperset()
Report whether this set contains another set.

pop()
Remove and return an arbitrary set element. Raises KeyError if the set is empty.

remove()
Remove an element from a set; it must be a member.

If the element is not a member, raise a KeyError.

symmetric_difference()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

symmetric_difference_update()
Update a set with the symmetric difference of itself and another.

union()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

update()
Update a set with the union of itself and others.

class skeletor.core.compat.**unicode**
unicode(object='') -> unicode object unicode(string[, encoding[, errors]]) -> unicode object

Create a new Unicode object from the given encoded string. encoding defaults to the current default string encoding. errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.

capitalize() → unicode
Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

center(*width*[, *fillchar*]) → unicode
Return S centered in a Unicode string of length width. Padding is done using the specified fill character (default is a space)

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in Unicode string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode([*encoding*[, *errors*]]) → string or unicode
Decodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that is able to handle UnicodeDecodeErrors.

encode([*encoding*[, *errors*]]) → string or unicode
Encodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs([*tabsize*]) → unicode
Return a copy of S where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → unicode
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Like S.find() but raise ValueError when the substring is not found.

isalnum() → bool
Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha() → bool
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal() → bool
Return True if there are only decimal characters in S, False otherwise.

isdigit() → bool
Return True if all characters in S are digits and there is at least one character in S, False otherwise.

islower() → bool
Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric() → bool
Return True if there are only numeric characters in S, False otherwise.

isspace() → bool
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle() → bool
Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper() → bool
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join(*iterable*) → unicode
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust(*width*[, *fillchar*]) → int
Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower() → unicode
Return a copy of the string S converted to lowercase.

lstrip([*chars*]) → unicode
Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

partition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace(*old*, *new*[, *count*]) → unicode
Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Like S.rfind() but raise ValueError when the substring is not found.

rjust(*width*[, *fillchar*]) → unicode
Return S right-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

rpartition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip([*chars*]) → unicode
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

split([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines(*keepends=False*) → list of strings
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip([*chars*]) → unicode
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

swapcase() → unicode
Return a copy of S with uppercase characters converted to lowercase and vice versa.

title() → unicode
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

translate(*table*) → unicode
Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted.

upper() → unicode
Return a copy of S converted to uppercase.

zfill(*width*) → unicode
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

class skeletor.core.compat.**long**
long(x=0) -> long long(x, base=10) -> long

Convert a number or string to a long integer, or return 0L if no arguments are given. If x is floating point, the conversion truncates towards zero.

If x is not a number or if base is given, then x must be a string or Unicode object representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4L

bit_length() → int or long
Number of bits necessary to represent self in binary. >>> bin(37L) '0b100101' >>> (37L).bit_length() 6

conjugate()
Returns self, the complex conjugate of any long.

denominator
the denominator of a rational number in lowest terms

imag
the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

skeletor.core.compat.**unichr**(*i*) → Unicode character
Return a Unicode string of one character with ordinal i; 0 <= i <= 0x10ffff.

skeletor.core.json – JSON utilities

skeletor.core.json.**dumps**(*obj*, ***kwargs*)
Serialize an object into a JSON string

skeletor.core.json.**loads**(*json_str*)
Load an object from a JSON string

skeletor.core.json.**read**(*path*)
Read JSON objects; return an empty dict on error

skeletor.core.json.**write**(*obj*, *path*)
Write object as JSON to path

creates directories if needed

skeletor.core.log – Logging

class skeletor.core.log.**ConsoleHandler**(*combine_stderr=False*)
Pass logging.INFO to stdout, everything else to stderr

emit(*record*)

flush()

class skeletor.core.log.**Formatter**(*fmt='%(levelno)s: %(msg)s'*)
Custom formatter to allow a custom output format per level

debug_fmt = 'debug: %(module)s: %(lineno)d: %(msg)s'

error_fmt = 'error: %(message)s'

format(*record*)

info_fmt = '%(message)s'

warn_fmt = 'warning: %(message)s'

class skeletor.core.log.**Logger**(*name*)

json(*obj*, ***kwargs*)

pprint(*obj*)

trace(*msg*, *depth=1*)

traceback()

class skeletor.core.log.**TextDecorator**(*force=False*)

BOLD = '\x1b[1;1m'

GREEN = '\x1b[0;32m'

RED = '\x1b[0;31m'

RESET = '\x1b[0m'

YELLOW = '\x1b[0;33m'

ansi(*code*, *text*)

bold(*text*)

green(*text*)

red(*text*)

yellow(*text*)

skeletor.core.log.**decorator**(*force=False*)
Return a text decorator

skeletor.core.log.**init**(*verbose*, *combine_stderr=False*)
Initialize logging

Parameters **verbose** – Enables debugging output

skeletor.core.log.**logger**(*name=None*)
Return a module-scope logger

skeletor.core.util – Utilities

skeletor.core.util.**deep_update**(*a*, *b*)
Allow piece-wise overriding of dictionaries

skeletor.core.util.**import_string**(*modstr*)
Resolve a package.module.variable string

skeletor.core.util – Library version

3.1.2 skeletor.util – Reusable utilities

skeletor.util.config – Configuration file readers

class skeletor.util.config.**Config**(*path*, *environ=None*)
Read configuration values from a file

initialize_environ()

read()

reset()

value(*key*, *env=None*)
Return a configured value

class skeletor.util.config.**JSONConfig**(*path*, *environ=None*)
Read configuration values from a JSON file

read()

skeletor.util.decorators – Function decorators for managing contexts

Decorators to simplify resource allocation

Many methods will require access to an initialized resource. The `acquire_context()` decorator provides a live resource context to its decorated function as the first argument.

Callers can override the decorator-provided resource by passing in *context*=... as a keyword argument at the call site, which is needed when the resource has been pre-allocated and needs to be reused between calls.

class skeletor.util.decorators.**Context**
Base class for custom contexts

acquire()
Called once iff the context is constructed by the context manager

error()
Called when a exiting a context via an exception

release()
Called at the end of a context iff constructed by a manager

success()
Called when exiting a context successfully

class skeletor.util.decorators.**DefaultContextManager**(*context=None*, *default_factory=None*, **args*, ***kwargs*)

class skeletor.util.decorators.**DefaultFactory**
Creates contexts

static create(***kwargs*)

static filter_kwargs(*kwargs*)
Filter context arguments out from the function's kwargs

class skeletor.util.decorators.**acquire_context**(*default_factory*, *default_contextmgr=None*, *decorator=None*, **args*, ***kwargs*)
A decorator to provide methods with a live resource context

This decorator takes arguments, and thus the only time the to-be-decorated method is available is post-construction during the decoration process. This happens in __call__, and it is called once.

Methods inside a class can pass "decorator=staticmethod" if they want a static method.

The blind *args* and *kwargs* are passed to the *default_factory* when contexts are constructed.

class skeletor.util.decorators.**bind**(**args*, ***kwargs*)
A decorator to bind function parameters

class skeletor.util.decorators.**bindfunc**(*decorator=<function passthrough_decorator at 0x7f5d59bb9b90>*)
Allows fn.bind(foo=bar) when decorated on a decorator

This is a decorator decorator, which takes a decorator as input and returns another decorator when applied, and grants decorators the ability to do *mydecorator.bind(foo=bar)* and have it work without having to repeat *foo=bar* everywhere.

bind(**args*, ***kwargs*)
bind() stashes arguments

skeletor.util.decorators.**passthrough_decorator**(*f*)
Return a function as-is, unchanged

This is the default decorator used when creating contexts.

skeletor.util.string – String utilities

Functions for formatting various data strings

skeletor.util.string.**decode**(*string*)
decode(encoded_string) returns an unencoded unicode string

skeletor.util.string.**encode**(*string*)
encode(unencoded_string) returns a string encoded in utf-8

skeletor.util.string.**expand_path**(*path*, *environ=None*)
Resolve $VARIABLE and ~user paths

skeletor.util.string.**expandvars**(*path*, *environ=None*)
Like os.path.expandvars, but operates on a provided dictionary

os.environ is used when *environ* is None.

```
>>> expandvars('$foo', environ={'foo': 'bar'}) == 'bar'
True
```

```
>>> expandvars('$foo:$bar:${foo}${bar}', environ={'foo': 'a', 'bar': 'b'})  == 'a:b:ab'
True
```

3.1.3 skeletor.db – SQLAlchemy powertools

skeletor.db.context – Database contexts

class skeletor.db.context.**DatabaseContext**(*creator=None*, *context=None*, *commit=False*)

acquire()

commit()

error()

release()

rollback()

success()

class skeletor.db.context.**DatabaseFactory**

static create(*creator=None*, *context=None*, *commit=False*)

static filter_kwargs(*kwargs*)

skeletor.db.decorators – Decorators for providing contexts

Decorators to simplify database access

`skeletor.db.schema` – Schema definitions

`skeletor.db.sql` – SQLAlchemy helpers and utilities

`skeletor.db.table` – Query SQLAlchemy tables

CHAPTER 4

Contributing

4.1 Contributing to skeletor

We welcome contributions from everyone. Please fork this project on github.

4.1.1 Get the Code

```
git clone git://github.com/davvid/skeletor.git
```

4.1.2 Run the Test Suite

All tests must pass before code can be pulled into the master branch. If you are contributing an addition or a change in behavior, we ask that you document the change in the form of test cases.

In order to run the test cases you will need to install some dependencies into your test environment. We recomment using tox To simplify the process of setting up a test environment.

To run the suite, simply invoke *make test* or use tox to run the unittests across multiple Python versions:

```
$ make test
nosetests --with-doctest  skeletor tests
.............................
----------------------------------------------------------------------
Ran 29 tests in 0.544s
```

4.1.3 Generate Documentation

You do not need to generate the documentation when contributing, though, if you are interested, you can generate the docs yourself. The following requires Sphinx:

```
cd docs
make html
```

If you wish to browse the documentation, use Python's `SimpleHTTPServer` to host them at http://localhost:8000:

```
cd build/html
python -m SimpleHTTPServer
```

CHAPTER 5

Contact

Follow the project on Github, submit pull requests, and file issues. Please try to report bugs in the form of a test case that can be added to the test suite.

CHAPTER 6

Authors

- David Aguilar davvid -at- gmail.com - https://github.com/davvid

CHAPTER 7

Change Log

7.1 Change Log

7.1.1 Version 0.0.1 - January 2015

- The initial skeletor release contains powerful decorators and utility classes for use with SQLAlchemy databases.
- The core things needed to support a databse-backed application are configuration, logging, and database context/session management.
- Configuration is used for getting credentials since they should not live alongside the library and application code.
- Logging is needed for reporting database and application errors.
- Database context management is needed to streamline the work needed to establish database connections and transactions.
- Skeletor's database decorators allow you to tag a function as a "mutator", which means that calling it without an existing context will create a context, start a transation, and call into the function. If the function raises any exceptions then the transaction is automatically rolled back by the database context manager.

CHAPTER 8

License

skeletor is provided under a New BSD license,

S

A

B

C

D

E

F

G

I

J

L

N

P

R

S

T

U

V

W

Y

Z