xoutil Documentation

Release 1.8.0

Merchise

Oct 31, 2017

Contents

Collection of disparate utilities.

xoutil is essentially an extension to the Python's standard library, it does not turn into a full framework, but it's very useful to be used from a diversity of scenarios where compatibility is an important issue.

CHAPTER 1

What's new in 1.8.0

- Remove deprecated `xoutil.objects.metaclass`, use *`xoutil.eight.meta.metaclass()`* instead.
- Several modules are migrated to *`xoutil.future`*:
  - *`types`*.
  - *`collections`*.
  - *`datetime`*.
  - *`functools`*.
  - *`inspect`*.
  - *`codecs`*.
  - *`json`*.
  - *`threading`*.
  - *`subprocess`*.
  - *`pprint`*.
  - *`textwrap`*.
- Add function *`xoutil.deprecation.import_deprecated()`*, *`inject_deprecated()`* can be deprecated now.
- Add function `xoutil.deprecation.deprecate_linked()` to deprecate full modules imported from a linked version. The main example are all sub-modules of *`xoutil.future`*.
- Add function `xoutil.deprecation.deprecate_module()` to deprecate full modules when imported.
- Remove the module `xoutil.string` in favor of:
  - *`xoutil.future.codecs`*: Moved here functions *`force_encoding()`*, *`safe_decode()`*, and *`safe_encode()`*.
  - *`xoutil.eight.string`*: Technical string handling. In this module:

      * *force()*: Replaces old safe_str, and force_str versions.
      * *safe_join()*: Replaces old version in future module. This function is useless, it's equivalent to:

        ```
        force(vale).join(force(item) for item in iterator)
        ```

      * *force_ascii()*: Replaces old normalize_ascii. This function is safe and the result will be of standard str type containing only equivalent ASCII characters from the argument.

    – *xoutil.eight.text*: Text handling, strings can be part of internationalization processes. In this module:

      * *force()*: Replaces old safe_str, and force_str versions, but always returning the text type.
      * *safe_join()*: Replaces old version in future module, but in this case always return the text type. This function is useless, it's equivalent to:

        ```
        force(vale).join(force(item) for item in iterator)
        ```

    – capitalize_word function was completely removed, use instead standard method word.capitalize().
    – Functions capitalize, normalize_name, normalize_title, normalize_str, parse_boolean, parse_url_int were completely removed.
    – normalize_unicode was completely removed, it's now replaced by *xoutil.eight.text.force()*.
    – hyphen_name was moved to *xoutil.cli.tools*.
    – strfnumber was moved as an internal function of 'xoutil.future.datetime':mod: module.
    – Function normalize_slug is now deprecated. You should use now *slugify()*.

• Create __small__ protocol for small string representations, see xoutil.string.small() for more information.

• Remove xoutil.connote that was introduced provisionally in 1.7.1.

• Module *xoutil.params* was introduced provisionally in 1.7.1, but now has been fully recovered.

    – Add function *issue_9137()* – Helper to fix issue 9137 (self ambiguity).
    – Add function *check_count()* – Checker for positional arguments actual count against constrains.
    – Add function *check_default()* – Default value getter when passed as a last excess positional argument.
    – Add function *single()* – Return true only when a unique argument is given.
    – Add function *keywords_only()* – Decorator to make a function to accepts its keywords arguments as keywords-only.
    – Add function *pop_keyword_arg()* – Tool to get a value from keyword arguments using several possible names.
    – Add class *ParamManager* – Parameter manager in a "smart" way.
    – Add class *ParamScheme* – Parameter scheme definition for a manager.
    – Add class *ParamSchemeRow* – Parameter scheme complement.
    – Remove xoutil.params.ParamConformer.

- Module *`xoutil.values`* was recovered adding several new features (old name `xoutil.cl` was deprecated).
- Add **experimental** module *`xoutil.fp`* for Functional Programming stuffs.
- Add **experimental** module `xoutil.tasking`.
- Remove deprecated module `xoutil.data`. Add `xoutil.objects.adapt_exception()`.
- Remove deprecated `xoutil.dim.meta.Signature.isunit()`.

CHAPTER 2

Contents

`xoutil` – Collection of tools. Top-level imports

`xoutil.`**`Unset`** **= Unset**
False value, mainly for function parameter definitions, where None could be a valid value.

`xoutil.`**`Undefined`** **= Undefined**
False value for local scope use or where `Unset` could be a valid value

`xoutil.`**`Ignored`** **= Ignored**
To be used in arguments that are currently ignored because they are being deprecated. The only valid reason to use *Ignored* is to signal ignored arguments in method's/function's signature

`xoutil.annotate` - Py3k compatible annotations for Python 2

Provides Python 3k forward-compatible (**PEP 3107**) annotations.

Note: The *signature* argument for the `annotate()` in this module may not work on other python implementations than CPython. Currently, Pypy passes all but local variable tests.

`xoutil.annotate.`**`annotate`**(*signature=None*, ***annotations*)
Annotates a function with a Python 3k forward-compatible `__annotations__` mapping.

See **PEP 3107** for more details about annotations.

Parameters

- **`signature`** –

  A string with the annotated signature of the decorated function.

  This string should follow the annotations syntax in **PEP 3107**. But there are several deviations from the PEP text:

  – There's no support for the full syntax of Python 2 expressions; in particular nested arguments are not supported since they are deprecated and are not valid in Py3k.
  – Specifying defaults is no supported (nor needed). Defaults are placed in the signature of the function.
  – In the string it makes no sense to put an argument without an annotation, so this will raise an exception (SyntaxError).

- **`keyword_annotations`** – These are each mapped to a single annotation.

  Since you can't include the 'return' keyword argument for the annotation related with the return of the function, we provide several alternatives: if any of the following keywords arguments is provided (tested in the given order): 'return_annotation', '_return', '__return'; then it will be considered the 'return' annotation, the rest will be regarded as other annotations.

In any of the previous cases, you may provide more (or less) annotations than possible by following the PEP syntax. This is not considered an error, since the PEP allows annotations to be modified by others means.

If you provide a signature string **and** keywords annotations, the keywords will take precedence over the signature:

```
>>> @annotate('() -> list', return_annotation=tuple)
... def otherfunction():
...     pass

>>> otherfunction.__annotations__.get('return') is tuple
True
```

When parsing the *signature* the locals and globals in the context of the declaration are taken into account:

```
>>> interface = object # let's mock of ourselves
>>> class ISomething(interface):
...     pass

>>> @annotate('(a: ISomething) -> ISomething')
... def somewhat(a):
...     return a

>>> somewhat.__annotations__.get('a')
<class '...ISomething'>
```

`xoutil.bases` - Numeric base 32 and base 64 integer representations

Integer encoding and decoding in different bases.

`xoutil.bases.`**`int2str`**`(`*number*, *base=62*`)`

Return the string representation of an integer using a base.

Parameters **`base`** (*`Either an integer or a string with a custom table.`*) – The base.

Examples:

```
>>> int2str(65535, 16)
'ffff'

>>> int2str(65535)
'h31'

>>> int2str(65110208921, 'merchise')
'ehimseiemsce'

>>> int2str(651102, 2)
'10011110111101011110'
```

`xoutil.bases.`**`str2int`**`(`*src*, *base=62*`)`

Return the integer decoded from a string representation using a base.

Parameters **`base`** (*`Either an integer or a string with a custom table.`*) – The base.

Examples:

```
>>> str2int('ffff', 16)
65535

>>> str2int('1c', 16) == int('1c', 16)
True

>>> base = 'merchise'
>>> number = 65110208921
>>> str2int(int2str(number, base), base) == number
False

>>> base = 32
>>> str2int(int2str(number, base), base) == number
True
```

class `xoutil.bases.`**`B32`**

Handles base-32 conversions.

In base 32, each 5-bits chunks are represented by a single "digit". Digits comprises all symbols in 0..9 and a..v.

```
>>> B32.inttobase(32) == '10'
True
```

```
>>> B32.basetoint('10')
32
```

class `xoutil.bases.`**`B64`**

Handles [a kind of] base 64 conversions.

This **is not standard base64**, but a reference-friendly base 64 to help the use case of generating a short reference.

In base 64, each 6-bits chunks are represented by a single "digit". Digits comprises all symbols in 0..9, a..z, A..Z and the three symbols: *()[*.

```
>>> B64.inttobase(64) == '10'
True
```

```
>>> B64.basetoint('10')
64
```

Warning: In this base, letters **are** case sensitive:

```
>>> B64.basetoint('a')
10

>>> B64.basetoint('A')
36
```

`xoutil.bound` – Helpers for bounded execution of co-routines

New in version 1.6.3.

A bounded execution model

Some features are easy to implement using a generator or co-routine (**PEP 342**). For instance, you might want to "report units of work" one at a time. These kind of features could be easily programmed without any *bounds* whatsoever, and then you might "weave" the bounds.

This module helps to separate the work-doing function from the boundary-tests definitions.

This document uses the following terminology:

unbounded function This is the function that does the actual work without testing for any *boundary condition*. Boundary conditions are not "natural causes" of termination for the algorithm but conditions imposed elsewhere: the environment, resource management, etc.

This function *must* return a generator, called the *unbounded generator*.

unbounded generator The generator returned by an *unbounded function*. This generator is allowed to yield forever, although it could terminate by itself. So this is actually a *possibly* unbounded generator, but we keep the term to emphasize.

boundary condition It's a condition that does not belong to the logical description of any algorithm. When this condition is met it indicates that the *unbounded generator* should be closed. The boundary condition is tested each time the unbounded generator yields.

A boundary condition is usually implemented in a single function called the *boundary definition*.

boundary definition A function that implements a boundary condition. This function must comply with the boundary protocol (see `boundary()`).

Sometimes we identify the boundary condition with its *boundary definition*.

bounded function It's the result of applying a *boundary definition* to an *unbounded function*.

bounded generator It's the result of applying a *boundary condition* to an *unbounded generator*.

The bounded execution model takes at least an *unbounded generator* and a *boundary condition*. Applying the boundary condition to the unbounded generator ultimately results in a *bounded generator*, which will behave almost equivalently to the *unbounded generator* but will stop when the boundary condition yields True or when the unbounded generator itself is exhausted.

Included boundary conditions

`xoutil.bound.`**`timed`**`(`*maxtime*`)`

Becomes True after a given amount of time.

The bounded generator will be allowed to yields values until the *maxtime* time frame has elapsed.

Usage:

```
@timed(timedelta(seconds=60))
def do_something_in_about_60s():
    while True:
        yield
```

Note: This is a very soft limit.

We can't actually guarrant any enforcement of the time limit. If the bounded generator takes too much time or never yields this predicated can't do much. This usually helps with batch processing that must not exceed (by too much) a given amount of time.

The timer starts just after the `next()` function has been called for the predicate initialization. So if the *max-time* given is too short this predicated might halt the execution of the bounded function without allowing any processing at all.

If *maxtime* is not a timedelta, the timedelta will be computed as `timedelta(seconds=maxtime)`.

`xoutil.bound.`**`times`**`(`*n*`)`

Becomes True after a given after the *nth* item have been produced.

`xoutil.bound.`**`accumulated`**`(`*mass*, **attrs*, *initial=0*`)`

Becomes True after accumulating a given "mass".

mass is the maximum allowed to accumulate. This is usually a positive number. Each value produced by the *unbounded generator* is added together. Yield True when this amount to more than the given *mass*.

If any *attrs* are provided, they will be considered attributes (or keys) to search inside the yielded data from the bounded function. If no *attrs* are provided the whole data is accumulated, so it must allow addition. The attribute to be summed is extracted with *`get_first_of()`*, so only the first attribute found is added.

If the keyword argument *initial* is provided the accumulator is initialized with that value. By default this is 0.

`xoutil.bound.`**`pred`**`(`*func*, *skipargs=True*`)`

Allow "normal" functions to engage within the boundary protocol.

func should take a single argument and return True if the boundary condition has been met.

If *skipargs* is True then function *func* will not be called with the tuple `(args, kwargs)` upon initialization of the boundary, in that case only yielded values from the *unbounded generator* are passed. If you need to get the original arguments, set *skipargs* to False, in this case the first time *func* is called will be passed a single argument `(arg, kwargs)`.

Example:

```
>>> @pred(lambda x: x > 10)
... def fibonacci():
...     a, b = 1, 1
...     while True:
...        yield a
...        a, b = b, a + b
```

```
>>> fibonacci()
13
```

xoutil.bound.**until_errors**(**errors*)

Becomes True after any of *errors* has been raised.

Any other exceptions (except GeneratorExit) is propagated. You must pass at least an error.

Normally this will allow some possibly long jobs to be interrupted (SoftTimeLimitException in celery task, for instance) but leave some time for the caller to clean up things.

It's assumed that your job can be properly *finalized* after any of the given exceptions has been raised.

Parameters **on_error** – A callable that will only be called if the boundary condition is ever met, i.e if any of *errors* was raised. The callback is called before yielding True.

New in version 1.7.2.

Changed in version 1.7.5: Added the keyword argument *on_error*.

xoutil.bound.**until**(*time=None*, *times=None*, *errors=None*)

An idiomatic alias to other boundary definitions.

- `until(maxtime=n)` is the same as `timed(n)`.
- `until(times=n)` is the same as `times(n)`.
- `until(pred=func, skipargs=skip)` is the same as `pred(func, skipargs=skip)`.
- `until(errors=errors, **kwargs)` is the same as `until_errors(*errors, **kwargs)`.
- **`until(accumulate=mass, path=path, initial=initial)` is the same as** `accumulated(mass, *path.split('.'), initial=initial)`

Warning: You cannot mix many calls.

New in version 1.7.2.

Chaining several boundary conditions

To created a more complex boundary than the one provided by a single condition you could use the following high-level boundaries:

xoutil.bound.**whenany**(**boundaries*)

An OR-like boundary condition.

It takes several boundaries and returns a single one that behaves like the logical OR, i.e, will yield True when **any** of its subordinate boundary conditions yield True.

Calls `close()` of all subordinates upon termination.

Each *boundary* should be either:

- A "bare" boundary definition that takes no arguments.
- A boundary condition (i.e an instance of `BoundaryCondition`). This is result of calling a boundary definition.
- A generator object that complies with the boundary protocol. This cannot be tested upfront, a misbehaving generator will cause a RuntimeError if a boundary protocol rule is not followed.

Any other type is a TypeError.

`xoutil.bound.`**`whenall`**`(`*`*boundaries`*`)`

An AND-like boundary condition.

It takes several boundaries and returns a single one that behaves like the logical AND i.e, will yield True when **all** of its subordinate boundary conditions have yielded True.

It ensures that once a subordinate yields True it won't be sent more data, no matter if other subordinates keep on running and consuming data.

Calls `close()` of all subordinates upon termination.

Each *boundary* should be either:

- A "bare" boundary definition that takes no arguments.
- A boundary condition (i.e an instance of *`BoundaryCondition`*). This is result of calling a boundary definition.
- A generator object that complies with the boundary protocol. This cannot be tested upfront, a misbehaving generator will cause a RuntimeError if a boundary protocol rule is not followed.

Any other type is a TypeError.

Defining boundaries

If none of the boundaries defined deals with a boundary condition you have, you may create another one using *`boundary()`*. This is usually employed as decorator on the *boundary definition*.

`xoutil.bound.`**`boundary`**`(`*`definition`*`)`

Helper to define a boundary condition.

The *definition* must be a function that returns a generator. The following rules **must be** followed. Collectively these rules are called the *boundary protocol*.

- The *boundary definition* will yield True when and only when the boundary condition is met. Only the value True will signal the boundary condition.
- The *boundary definition* must yield at least 2 times:
  - First it will be called its `next()` method to allow for initialization of internal state.
  - Immediately after, it will be called its `send()` passing the tuple `(args, kwargs)` with the arguments passed to the *unbounded function*. At this point the boundary definition may yield True to halt the execution. In this case, the *unbounded generator* won't be asked for any value.
- The *boundary definition* must yield True before terminating with a StopIteration. For instance the following definition is invalid cause it ends without yielding True:

```
@boundary
def invalid():
    yield
    yield False
```

- The *boundary definition* must deal with GeneratorExit exceptions properly since we call the `close()` method of the generator upon termination. Termination occurs when the *unbounded generator* stops by any means, even when the boundary condition yielded True or the generator itself is exhausted or there's an error in the generator.

  Both *`whenall()`* and *`whenany()`* call the `close()` method of all their subordinate boundary conditions.

> Most of the time this reduces to *not* catching GeneratorExit exceptions.

A RuntimeError may happen if any of these rules is not followed by the *definition*. Furthermore, this error will occur when invoking the *bounded function* and not when applying the boundary to the *unbounded generator*.

Illustration of a boundary

Let's explain in detail the implementation of *times()* as an example of how a boundary condition could be implemented.

```
1 @boundary
2 def times(n):
3     '''Becomes True after the `nth` item have been produced.'''
4     passed = 0
5     yield False
6     while passed < n:
7         yield False
8         passed += 1
9     yield True
```

We implemented the boundary condition via the *boundary()* helper. This helpers allows to implement the boundary condition via a boundary definition (the function above). The `boundary` helper takes the definition and builds a *BoundaryCondition* instance. This instance can then be used to decorate the *unbounded function*, returning a *bounded function* (a *Bounded* instance).

When the *bounded function* is called, what actually happens is that:

- First the boundary condition is invoked passing the `n` argument, and thus we obtain the generator from the `times` function.
- We also get the generator from the unbounded function.
- Then we call `next(boundary)` to allow the `times` boundary to initialize itself. This runs the code of the `times` definition up to the line 5 (the first `yield` statement).
- The *bounded function* ignores the message from the boundary at this point.
- Then it sends the arguments passed to original function via the `send()` method of the boundary condition generator.
- This unfreezes the boundary condition that now tests whether `passes` is less that `n`. If this is true, the boundary yields False and suspends there at line 7.
- The *bounded function* see that message is not True and asks the *unbounded generator* for its next value.
- Then it sends that value to the boundary condition generator, which resumes execution at line 8. The value sent is ignored and `passes` gets incremented by 1.
- Again the generator asks if `passes` is less that `n`. If passes has reached `n`, it will execute line 9, yielding True.
- The *bounded function* see that the boundary condition is True and calls the `close()` method to the boundary condition generator.
- This is like raising a GeneratorExit just after resuming the `times` below line 9. The error is not trapped and propagates the `close()` method of the generator knows this means the generator has properly finished.

Note: Other boundaries might need to deal with GeneratorExit explicitly.

- Then the *bounded function* regains control and calls the `close()` method of the *unbounded generator*, this effectively raises a GeneratorExit inside the unbounded generator, which if untreated means everything went well.

If you look at the implementation of the *included boundary conditions*, you'll see that all have the same pattern:

1. Initialization code, followed by a `yield False` statement. This is a clear indicator that the included boundary conditions disregard the first message (the arguments to the unbounded function).
2. A looping structure that tests the condition has not been met and yields False at each cycle.
3. The `yield True` statement outside the loop to indicate the boundary condition has been met.

This pattern is not an accident. Exceptionally `whenall()` and `whenany()` lack the first standalone *yield False* because they must not assume all its subordinate predicates will ignore the first message.

Internal API

class `xoutil.bound.`**`Bounded`**`(`*target*`)`

The bounded function.

This is the result of applying a *boundary definition* to an *unbounded function* (or generator).

If *target* is a function this instance can be called several times. If it's a generator then it will be closed after either calling (`__call__`) this instance, or consuming the generator given by `generate()`.

This class is actually subclassed inside the `apply()` so that the weaving boundary definition with the *target* unbounded function is not exposed.

`__call__``(`**args*, ***kwargs*`)`

Return the last value from the underlying *bounded generator*.

`generate``(`**args*, ***kwargs*`)`

Return the *bounded generator*.

This method exposes the *bounded generator*. This allows you to "see" all the values yielded by the *unbounded generator* up to the point when the boundary condition is met.

class `xoutil.bound.`**`BoundaryCondition`**`(`*definition*, *name=None*, *errors=None*`)`

Embodies the boundary protocol.

The *definition* argument must a function that implements a *boundary definition*. This function may take arguments to initialize the state of the boundary condition.

Instances are callables that will return a `Bounded` subclass specialized with the application of the *boundary condition* to a given unbounded function (*target*). For instance, `times(6)` returns a class, that when instantiated with a *target* represents the bounded function that takes the 6th valued yielded by target.

If the *definition* takes no arguments for initialization you may pass the *target* directly. This is means that if `__call__()` receives arguments they will be used to instantiate the `Bounded` subclass, ie. this case allows only a single argument *target*.

If *errors* is not None it should be a tuple of exceptions to catch and throw inside the boundary condition definition. Other exceptions, beside GeneratorExit and StopIteration, are not handled (so the bubble up). See `until_error()`.

An example: time bounded batch processing

We have a project in which we need to send emails inside a *cron* task (celery is not available). Emails to be sent are placed inside an *Outbox* but we may only spent about 60 seconds to send as many emails as we can. If our emails are reasonably small (i.e will be delivered to the SMTP server in a few miliseconds) we could use the `timed()` predicate to bound the execution of the task:

```
@timed(50)
def send_emails():
   outbox = Outbox.open()
   try:
      for message in outbox:
         emailbackend.send(message)
         outbox.remove(message)
         yield message
   except GeneratorExit:
      # This means the time we were given is off.
      pass
   finally:
      outbox.close()   # commit the changes to the outbox
```

Notice that you **must** enclose your batch-processing code in a `try` statement if you need to somehow commit changes. Since we may call the `close()` method of the generator to signal that it must stop.

A `finally` clause is not always appropriated cause an error that is not GeneratorExit error should not commit the data unless you're sure data changes that were made before the error could be produced. In the code above the only place in the code above where an error could happen is the sending of the email, and the data is only touched for each email that is actually sent. So we can safely close our outbox and commit the removal of previous message from the outbox.

Using the `Bounded.generate()` method

Calling a *bounded generator* simply returns the last valued produced by the *unbounded generator*, but sometimes you need to actually *see* all the values produced. This is useful if you need to meld several *generators* with partially overlapping boundary conditions.

Let's give an example by extending a bit the example given in the previous section. Assume you now need to extend your cron task to also read an Inbox as much as it can and then send as many messages as it can. Both things should be done under a given amount of time, however the accumulated size of sent messages should not surpass a threshold of bytes to avoid congestion.

For this task you may use both `timed()` and `accumulated()`. But you must apply `accumulated()` only to the process of sending the messages and the *timed* boundary to the overall process.

This can be accomplished like this:

```
def communicate(interval, bandwidth):
    from itertools import chain as meld

    def receive():
        for message in Inbox.receive():
           yield message

    @accumulated(bandwith, 'size')
    def send():
        for message in Outbox.messages():
            yield message

    @timed(interval)
    def execute():
        for _ in meld(receive(), send.generate()):
            yield
    return execute()
```

Let's break this into its parts:

- The `receive` function reads the Inbox and yields each message received.

  It is actually an *unbounded function* but don't want to bound its execution in isolation.

- The `send` unbounded function sends every message we have in the Outbox and yields each one. In this case we *can* apply the *accumulated* boundary to get a `Bounded` instance.

- Then we define an *execute* function bounded by *timed*. This function melds the `receive` and `send` processes, but we can't actually call `send` because we need to yield after each message has been received or sent. That's why we need to call the `generate()` so that the time boundary is also applied to the sending process.

Note: The structure from this example is actually taken from a real program, although simplified to serve better for learning. For instance, in our real-world program *bandwidth* could be None to indicate no size limit should be applied to the sending process. Also in the example we're not actually saving nor sending messages!

`xoutil.cli` – Command line application facilities

Tools for Command-Line Interface (CLI) applications.

CLI is a mean of interaction with a computer program where the user (or client) issues commands to the program in the form of successive lines of text (command lines).

Commands can be registered by:

- sub-classing the `Command`,
- using `register()` ABC mechanism for virtual sub-classes,
- redefining ~*'Command.sub_commands'* class method.

New in version 1.4.1.

class `xoutil.cli.`**`Command`**

Base for all commands.

classmethod **`cli_name`**`()`

Calculate the command name.

Standard method uses ~*xoutil.cli.tools.hyphen_name*. Redefine it to obtain a different behaviour.

Example:

```
>>> class MyCommand(Command):
...     pass

>>> MyCommand.cli_name() == 'my-command'
True
```

`run``(`*args=None*`)`

Must return a valid value for "sys.exit"

classmethod **`set_default_command`**`(`*cmd=None*`)`

Default command is called when no one is specified.

A command is detected when its name appears as the first command-line argument.

To specify a default command, use this method with the command as a string (the command name) or the command class.

If the command is specified, then the calling class is the selected one.

For example:

```
>>> Command.set_default_command('server')
>>> Server.set_default_command()
>>> Command.set_default_command(Server)
```

class `xoutil.cli.`**`Help`**

Show all commands.

Define the class attribute *__order__* to sort commands in special command "help".

Commands could define its help in the first line of a sequence of documentations until found:

- command class,
- "run" method,
- definition module.

This command could not be overwritten unless using the class attribute:

__override__ = True

classmethod **`get_arg_parser`**`()`

This is an example on how to build local argument parser.

Use class method "get

Applications

A simple `main()` entry point for CLI based applications.

This module provides an example of how to use `xoutil.cli` to create a CLI application.

`xoutil.cli.app.`**`main`**`(`*default=None*`)`

Execute a command.

It can be given as the first program argument or it's the *default* command is defined.

Tools

Utilities for command-line interface (CLI) applications.

- `program_name()`: calculate the program name from "sys.argv[0]".
- **`command_name()`**\ [calculate command names using class names in lower] case inserting a hyphen before each new capital letter.

`xoutil.cli.tools.`**`command_name`**`(`*cls*`)`

Calculate a command name from given class.

Names are calculated putting class names in lower case and inserting hyphens before each new capital letter. For example "MyCommand" will generate "my-command".

It's defined as an external function because a class method don't apply to minimal commands (those with only the "run" method).

Example:

```
>>> class SomeCommand(object):
...     pass

>>> command_name(SomeCommand) == 'some-command'
True
```

If the command class has an attribute *command_cli_name*, this will be used instead:

```
>>> class SomeCommand(object):
...    command_cli_name = 'adduser'

>>> command_name(SomeCommand) == 'adduser'
True
```

It's an error to have a non-string *command_cli_name* attribute:

```
>>> class SomeCommand(object):
...    command_cli_name = None

>>> command_name(SomeCommand)
Traceback (most recent call last):
   ...
TypeError: Attribute 'command_cli_name' must be a string.
```

xoutil.cli.tools.**hyphen_name**(*name*, *join_numbers=True*)
Convert a name to a hyphened slug.

Expects a *name* in Camel-Case. All invalid characters (those invalid in Python identifiers) are ignored. Numbers are joined with preceding part when *join_numbers* is True.

For example:

```
>>> hyphen_name('BaseNode') == 'base-node'
True

>> hyphen_name('--__ICQNámeP12_34Abc--') == 'icq-name-p12-34-abc'
True

>> hyphen_name('ICQNámeP12', join_numbers=False) == 'icq-name-p-12'
True
```

xoutil.cli.tools.**program_name**()
Calculate the program name from "sys.argv[0]".

xoutil.clipping - Text clipping and trimming

Text clipping and trimming.

xoutil.clipping.**DEFAULT_MAX_WIDTH = 64**
Default value for *max_width* parameter in functions that reduce strings, see crop() and small().

xoutil.clipping.**ELLIPSIS = '…'**
Value used as a fill when a string representation is brimmed over.

xoutil.clipping.**MIN_WIDTH = 8**
Value for *max_width* parameter in functions that reduce strings, must not be less than this value.

`xoutil.clipping.`**`crop`**`(`*obj*, *max_width=None*`)`

Return a reduced string representation of *obj*.

Classes can now define a new special method or attribute named '__crop__'.

If *max_width* is not given, defaults to `DEFAULT_MAX_WIDTH`.

New in version 1.8.0.

`xoutil.clipping.`**`crop_iterator`**`(`*obj*, *max_width=None*`)`

Return a reduced string representation of the iterator *obj*.

See `crop()` function for a more general tool.

If *max_width* is not given, defaults to `DEFAULT_MAX_WIDTH`.

New in version 1.8.0.

`xoutil.clipping.`**`small`**`(`*obj*, *max_width=None*`)`

Crop the string representation of *obj* and make some replacements.

- Lambda function representations ('<lambda>' by 'λ').
- Ellipsis ('...' by '…')

If max_width is not given, defaults to `DEFAULT_MAX_WIDTH`.

New in version 1.8.0.

`xoutil.context` - Simple execution contexts

A context manager for execution context flags.

`xoutil.context.`**`context`**

alias of `Context`

class `xoutil.context.`**`Context`**`(`**args*, ***kwargs*`)`

An execution context manager with parameters (or flags).

Use as:

```
>>> SOME_CONTEXT = object()
>>> from xoutil.context import context
>>> with context(SOME_CONTEXT):
...     if context[SOME_CONTEXT]:
...         print('In context SOME_CONTEXT')
In context SOME_CONTEXT
```

Note the difference creating the context and checking it: for entering a context you should use `context(name)` for testing whether some piece of code is being executed inside a context you should use `context[name]`; you may also use the syntax *name in context*.

When an existing context is re-enter, the former one is reused. Nevertheless, the data stored in each context is local to each level.

For example:

```
>>> with context('A', b=1) as a1:
...   with context('A', b=2) as a2:
...       print(a1 is a2)
...       print(a2['b'])
...   print(a1['b'])
```

```
True
2
1
```

For data access, a mapping interface is provided for all contexts. If a data slot is deleted at some level, upper level is used to read values. Each new written value is stored in current level without affecting upper levels.

For example:

```
>>> with context('A', b=1) as a1:
...   with context('A', b=2) as a2:
...       del a2['b']
...       print(a2['b'])
1
```

It is an error to *reuse* a context directly like in:

```
>>> with context('A', b=1) as a1:
...   with a1:
...       pass
Traceback (most recent call last):
...
RuntimeError: Entering the same context level twice! ...
```

Note: About thread-locals and contexts.

The `context` uses internally a `thread-local` instance to keep context stacks in different threads from seeing each other.

If, when this module is imported, `greenlet` is **imported** already, greenlet isolation is also warranted (which implies thread isolation).

If you use collaborative multi-tasking based in other framework other than *greenlet*, you must ensure to monkey patch the *threading.local* class so that isolation is kept.

In future releases of xoutil, we plan to provide a way to inject a "process" identity manager so that other frameworks be easily integrated.

Changed in version 1.7.1: Changed the test about `greenlet`. Instead of testing for *greenlet* to be importable, test it is imported already.

Changed in version 1.6.9: Added direct greenlet isolation and removed the need for `gevent.local`.

New in version 1.6.8: Uses `gevent.local` if available to isolate greenlets.

`xoutil.cpystack` - Utilities to inspect the CPython's stack

Utilities to inspect the CPython's stack.

`xoutil.cpystack.`**`getargvalues`**(*frame*)

Inspects the given frame for arguments and returns a dictionary that maps parameters names to arguments values. If an * argument was passed then the key on the returning dictionary would be formatted as *<name-of-*-param>[index]*.

For example in the function:

```
>>> def autocontained(a, limit, *margs, **ks):
...     import sys
...     return getargvalues(sys._getframe())

>>> autocontained(1, 12)['limit']
12

>>> autocontained(1, 2, -10, -11)['margs[0]']
-10
```

In Python 2.7, packed arguments also works:

```
>>> def nested((x, y), radius):
...     import sys
...     return getargvalues(sys._getframe())

>>> nested((1, 2), 12)['y']
2
```

`xoutil.cpystack.`**`error_info`**(**args*, ***kwargs*)

Get error information in current trace-back.

No all trace-back are returned, to select which are returned use:

- `args`: Positional parameters
  - If string, represent the name of a function.
  - If an integer, a trace-back level.

  Return all values.
- `kwargs`: The same as `args` but each value is a list of local names to return. If a value is `True`, means all local variables.

Return a list with a dict in each item.

Example:

```
>>> def foo(x):
...     x += 1//x
...     if x % 2:
...         bar(x - 1)
...     else:
...         bar(x - 2)

>>> def bar(x):
...     x -= 1//x
...     if x % 2:
...         foo(x//2)
...     else:
...         foo(x//3)

>>> try:
...     foo(20)
... except:
...     print(printable_error_info('Example', foo=['x'], bar=['x']))
Example
   ERROR: integer division or modulo by zero
   ...
```

xoutil.cpystack.**object_info_finder**(*obj_type*, *arg_name=None*, *max_deep=25*)

Find an object of the given type through all arguments in stack frames.

Returns a tuple with the following values: (arg-value, arg-name, deep, frame).

When no object is found None is returned.

Arguments: object_type: a type or a tuple of types as in "isinstance". arg_name: the arg_name to find; if None find in all arguments max_deep: the max deep to enter in the stack frames.

xoutil.cpystack.**object_finder**(*obj_type*, *arg_name=None*, *max_deep=25*)

Find an object of the given type through all arguments in stack frames.

The difference with `object_info_finder()` is that this function returns the object directly, not a tuple.

xoutil.cpystack.**track_value**(*value*, *max_deep=25*)

Find a value through all arguments in stack frames.

Returns a dictionary with the full-context in the same level as "value".

xoutil.cpystack.**iter_stack**(*max_deep=25*)

Iterates through stack frames until exhausted or *max_deep* is reached.

To find a frame fulfilling a condition use:

```
frame = next(f for f in iter_stack() if condition(f))
```

Using the previous pattern, functions *object_info_finder*, *object_finder* and *track_value* can be reprogrammed or deprecated.

New in version 1.6.8.

xoutil.cpystack.**iter_frames**(**args*, ***kw*)

Iterates through all stack frames.

Returns tuples with the following:

```
(deep, filename, line_no, start_line).
```

New in version 1.1.3.

Deprecated since version 1.6.8: The use of params *attr_filter* and *value_filter*.

`xoutil.crypto` - Other cryptographic services

General security tools.

Adds the ability to generate new passwords using a source pass-phrase and a secury strong level.

xoutil.crypto.**generate_password**(*pass_phrase*, *level=3*)

Generate a password from a source *pass-phrase* and a security *level*.

Parameters

- **pass_phrase** – String pass-phrase to be used as base of password generation process.
- **level** – Numerical security level (the bigger the more secure, but don't exaggerate!).

When *pass_phrase* is a valid string, *level* means a generation method. Each level implies all other with an inferior numerical value.

There are several definitions with numerical values for *level* (0-4):

PASS_PHRASE_LEVEL_BASIC

Generate the same pass-phrase, just removing invalid characters and converting the result to lower-case.

PASS_PHRASE_LEVEL_MAPPED

Replace some characters with new values: `'e'->'3'`, `'i'->'1'`, `'o'->'0'`, `'s'->'5'`.

PASS_PHRASE_LEVEL_MAPPED_MIXED

Consonants characters before 'M' (included) are converted to upper-case, all other are kept lower-case.

PASS_PHRASE_LEVEL_MAPPED_DATED

Adds a suffix with the year of current date ("<YYYY>").

PASS_PHRASE_LEVEL_STRICT

Randomly scramble previous result until unbreakable strong password is obtained.

If *pass_phrase* is `None` or an empty string, generate a "secure salt" (a password not based in a source pass-phrase). A "secure salt" is generated by scrambling the concatenation of a random phrases from the alphanumeric vocabulary.

Returned password size is `4*level` except when a *pass-phrase* is given for *level* <= 4 where depend on the count of valid characters of *pass-phrase* argument, although minimum required is warranted. When *pass-phrase* is `None` for *level* zero or negative, size `4` is assumed. First four levels are considered weak.

Maximum size is defined in the *MAX_PASSWORD_SIZE* constant.

Default level is *PASS_PHRASE_LEVEL_MAPPED_DATED* when using a pass-phrase.

`xoutil.crypto.`**PASS_PHRASE_LEVEL_BASIC = 0**
The most basic level (less) for the password generation.

`xoutil.crypto.`**PASS_PHRASE_LEVEL_MAPPED = 1**
A level for simply mapping of several chars.

`xoutil.crypto.`**PASS_PHRASE_LEVEL_MAPPED_MIXED = 2**
Another "stronger" mapping level.

`xoutil.crypto.`**PASS_PHRASE_LEVEL_MAPPED_DATED = 3**
Appends the year after mapping.

`xoutil.crypto.`**PASS_PHRASE_LEVEL_STRICT = 4**
Totally scramble the result, making very hard to predict the result.

`xoutil.crypto.`**DEFAULT_PASS_PHRASE_LEVEL = 3**
The default level for *generate_password()*

`xoutil.crypto.`**MAX_PASSWORD_SIZE = 512**
An upper limit for generated password length.

`xoutil.decorator` - Several decorators

This module contains several useful decorators, for several purposed. Also it severs as a namespace for other well-defined types of decorators.

Warning: This modules will be progressively deprecated during the 1.6 series.

We feel that either `xoutil.objects` or *xoutil.functools* are a better match for some of these decorators. But since we need to make sure about keeping dependencies, the deprecation won't be final until 1.7.0. After 1.8.0, this modules will be finally removed.

Top-level decorators

class `xoutil.decorator.`**`AttributeAlias`**(*attr_name*)

Descriptor to create aliases for object attributes.

This descriptor is mainly to be used internally by `aliases()` decorator.

`xoutil.decorator.`**`settle`**(***kwargs*)

Returns a decorator to settle attributes to the decorated target.

Usage:

```
>>> @settle(name='Name')
... class Person(object):
...     pass

>>> Person.name
'Name'
```

`xoutil.decorator.`**`namer`**(*name*, ***kwargs*)

Like `settle()`, but '__name__' is a required positional argument.

Usage:

```
>>> @namer('Identity', custom=1)
... class I(object):
...     pass

>>> I.__name__
'Identity'

>>> I.custom
1
```

`xoutil.decorator.`**`aliases`**(**names*, ***kwargs*)

In a class, create an `AttributeAlias` descriptor for each definition as keyword argument (alias=existing_attribute).

If "names" are given, then the definition context is looked and are assigned to it the same decorator target with all new names:

```
>>> @aliases('foo', 'bar')
... def foobar(*args):
...     'This function is added to its module with two new names.'
```

`xoutil.decorator.`**`assignment_operator`**(*func*, *maybe_inline=False*)

Makes a function that receives a name, and other args to get its first argument (the name) from an assignment operation, meaning that it if its used in a single assignment statement the name will be taken from the left part of the = operator.

Warning: This function is dependant of CPython's implementation of the language and won't probably work on other implementations. Use only you don't care about portability, but use sparingly (in case you change your mind about portability).

`xoutil.decorator.`**`instantiate`**(*target*, **args*, ***kwargs*)

Some singleton classes must be instantiated as part of its declaration because they represents singleton objects.

Every argument, positional or keyword, is passed as such when invoking the target. The following two code samples show two cases:

```
>>> @instantiate
... class Foobar(object):
...     def __init__(self):
...         print('Init...')
Init...


>>> @instantiate('test', context={'x': 1})
... class Foobar(object):
...     def __init__(self, name, context):
...         print('Initializing a Foobar instance with name={name!r} '
...               'and context={context!r}'.format(**locals()))
Initializing a Foobar instance with name='test' and context={'x': 1}
```

In all cases, Foobar remains the class, not the instance:

```
>>> Foobar
<class '...Foobar'>
```

class `xoutil.decorator.`**`memoized_property`**(*fget*, *doc=None*)

A read-only property that is only evaluated once.

This is extracted from the SQLAlchemy project's codebase, merit and copyright goes to SQLAlchemy authors:

```
Copyright (C) 2005-2011 the SQLAlchemy authors and contributors

This module is part of SQLAlchemy and is released under the MIT License:
http://www.opensource.org/licenses/mit-license.php
```

class `xoutil.decorator.`**`memoized_instancemethod`**(*fget*, *doc=None*)

Decorate a method memoize its return value.

Best applied to no-arg methods: memoization is not sensitive to argument values, and will always return the same value even when called with different arguments.

This is extracted from the SQLAlchemy project's codebase, merit and copyright goes to SQLAlchemy authors:

```
Copyright (C) 2005-2011 the SQLAlchemy authors and contributors

This module is part of SQLAlchemy and is released under the MIT License:
http://www.opensource.org/licenses/mit-license.php
```

Sub packages

`xoutil.decorator.development` - Decorators for development annotations

`xoutil.decorator.development.`**`unstable`**(*target*, *msg=None*)

Declares that a method, class or interface is unstable.

This has the side-effect of issuing a warning the first time the *target* is invoked.

The *msg* parameter, if given, should be string that contains, at most, two positional replacement fields ({0} and {1}). The first replacement field will be the type of *target* (interface, class or function) and the second matches *target's* full name.

`xoutil.decorator.meta` - Decorator-making facilities

Decorator-making facilities.

This module provides a signature-keeping version of the `xoutil.decorators.decorator()`, which is now deprecated in favor of this module's version.

We scinded the decorator-making facilities from decorators per se to allow the module `xoutil.deprecation` to be used by decorators and at the same time, implement the decorator `deprecated()` more easily.

This module is an adapted work from the decorator version 3.3.2 package and is copyright of its owner as stated below. Adaptation work is done by Merchise.

Original copyright and license notices from decorator package:

class `xoutil.decorator.meta.`**`FunctionMaker`**(*func=None*, *name=None*, *signature=None*, *defaults=None*, *doc=None*, *module=None*, *funcdict=None*)

An object with the ability to create functions with a given signature. It has attributes name, doc, module, signature, defaults, dict and methods update and make.

classmethod **`create`**(*obj*, *body*, *evaldict*, *defaults=None*, *doc=None*, *module=None*, *addsource=True*, ***attrs*)

Create a function from the strings name, signature and body. "evaldict" is the evaluation dictionary. If addsource is true an attribute __source__ is added to the result. The attributes attrs are added, if any.

`make`(*src_templ*, *evaldict=None*, *addsource=False*, ***attrs*)

Make a new function from a given template and update the signature

update (*func*, ***kw*)

Update the signature of func with the data in self

`xoutil.decorator.meta.`**flat_decorator** (*caller*, *func=None*)

Creates a signature keeping decorator.

`decorator(caller)` converts a caller function into a decorator.

`decorator(caller, func)` decorates a function using a caller.

`xoutil.decorator.meta.`**decorator** (*caller*)

Eases the creation of decorators with arguments. Normally a decorator with arguments needs three nested functions like this:

```
def decorator(*decorator_arguments):
    def real_decorator(target):
        def inner(*args, **kwargs):
            return target(*args, **kwargs)
        return inner
    return real_decorator
```

This decorator reduces the need of the first level by comprising both into a single function definition. However it does not removes the need for an `inner` function:

```
>>> @decorator
... def plus(target, value):
...     from functools import wraps
...     @wraps(target)
...     def inner(*args):
...         return target(*args) + value
...     return inner

>>> @plus(10)
... def ident(val):
...     return val

>>> ident(1)
11
```

A decorator with default values for all its arguments (except, of course, the first one which is the decorated *target*) may be invoked without parenthesis:

```
>>> @decorator
... def plus2(func, value=1, missing=2):
...     from functools import wraps
...     @wraps(func)
...     def inner(*args):
...         print(missing)
...         return func(*args) + value
...     return inner

>>> @plus2
... def ident2(val):
...     return val

>>> ident2(10)
2
11
```

But (if you like) you may place the parenthesis:

```
>>> @plus2()
... def ident3(val):
...     return val

>>> ident3(10)
2
11
```

However, this is not for free, you cannot pass a single positional argument which type is a function:

```
>>> def p():
...    print('This is p!!!')

>>> @plus2(p)
... def dummy():
...    print('This is dummy')
Traceback (most recent call last):
    ...
TypeError: p() takes ...
```

The workaround for this case is to use a keyword argument.

`xoutil.deprecation` - Utils for marking deprecated elements

`xoutil.deprecation.`**`deprecated`**(*replacement*, *msg=None*, *deprecated_module=None*, *removed_in_version=None*, *check_version=False*)

Small decorator for deprecated functions.

Usage:

```
@deprecated(new_function)
def deprecated_function(...):
    ...
```

Parameters

- **`replacement`** – Either a string or the object that replaces the deprecated.
- **`msg`** – A deprecation warning message template. You should provide keyword arguments for the `format()` function. Currently we pass the current keyword arguments: *replacement* (after some processing), *funcname* with the name of the currently deprecated object and *in_version* with the version this object is going to be removed if *removed_in_version* argument is not None.

  Defaults to: "{funcname} is now deprecated and it will be removed{in_version}. Use {replacement} instead."
- **`removed_in_version`** – The version the deprecated object is going to be removed.
- **`check_version`** – If True and *removed_in_version* is not None, then declarations of obseleted objects will raise a DeprecationError. This helps the release manager to keep the release clean.

Note: Currently only works with setuptools' installed distributions.

- **deprecated_module** – If provided, the name of the module the deprecated object resides. Not needed if the deprecated object is a function or class.
- **new_name** – If provided, it's used as the name of the deprecated object. Needed to allow renaming in *import_deprecated()* helper function.

Changed in version 1.4.1: Introduces removed_in_version and check_version.

xoutil.deprecation.**import_deprecated**(*module*, **names*, ***aliases*)

Import functions deprecating them in the target module.

The target module is the caller of this function (only intended to be called in the global part of a module).

Parameters

- **module** – The module from which functions will be imported. Could be a string, or an imported module.
- **names** – The names of the functions to import.
- **aliases** – Keys are the new names, values the old names.

For example:

```
>>> from xoutil.deprecation import import_deprecated
>>> import math
>>> import_deprecated(math, 'sin', new_cos='cos')
>>> sin is not math.sin
True
```

Next examples are all `True`, but them print the deprecation warning when executed:

```
>>> sin(math.pi/2) == 1.0
>>> new_cos(2*math.pi) == math.cos(2*math.pi)
```

If no identifier is given, it is assumed equivalent as `from module import *`.

The statement `import_deprecated('math', 'sin', new_cos='cos')` has the same semantics as `from math import sin, cos as new_cos`, but deprecating current module symbols.

This function is provided for easing the deprecation of whole modules and should not be used to do otherwise.

xoutil.deprecation.**inject_deprecated**(**args*, ***kw*)

Injects a set of functions from a module into another.

The functions will be marked as deprecated in the target module.

Parameters

- **funcnames** – function names to take from the source module.
- **source** – the module where the functions resides.
- **target** – the module that will contains the deprecated functions. If `None` will be the module calling this function.

This function is provided for easing the deprecation of whole modules and should not be used to do otherwise.

Deprecated since version 1.8.0: Use *import_deprecated()*.

`xoutil.dim` - Facilities to work with concrete numbers

The name 'dim' is a short of dimension. We borrow it from the topic dimensional analysis, even though the scope of this module is less ambitious.

This module is divided in two major parts: meta-definitions and applications.

`xoutil.dim.meta` – Meta-definitions for concrete numbers.

Facilities to work with concrete numbers.

> A concrete number is a number associated with the things being counted, in contrast to an abstract number which is a number as a single entity.
>
> —Wikipedia

This module allows you to define dimensions (or quantity types):

```
>>> from xoutil.dim.meta import Dimension, UNIT
>>> @Dimension.new
... class Length(object):
...     metre = UNIT
...     kilometre = 1000 * metre
...     centimetre = metre/100
...     milimetre = milimetres = metre/1000
...
...     inch = inches = 24.5 * milimetres
...     foot = feet = 12 * inches
```

See also:

Module *`base`* defines the standard base quantities.

Each dimension **must** define a single *canonical unit* for measuring quantities within the dimension. Values in the dimension are always expressed in terms of the canonical units.

In the previous example the dimension Length defined the *metre* for its canonical unit. The name of canonical unit defines the *`signature`* for the quantities in the dimension.

When printed (or `repr`-ed) *`quantities`* use the format `<magnitude>::<signature>`:

```
>>> metre = Length.metre
>>> metre
1::{<Length.metre>}/{}
```

Quantities support the standard arithmetical operations of addition, subtraction, multiplication and division. In fact, you obtain different quantities in the dimension by multiplying with the canonical unit:

```
>>> metre + metre
2::{<Length.metre>}/{}
```

```
>>> metre*metre
1::{<Length.metre>, <Length.metre>}/{}
```

```
>>> km = 1000 * metre
```

```
>>> 5 * km
5000::{<Length.metre>}/{}
```

Dimensional homogeneity imposes restrictions on the allowed operations between quantities. Only commensurable quantities (quantities of the same dimension) can be compared, equated, added, or subtracted.

```
>>> @Dimension.new
>>> class Time(object):
...     second = UNIT
```

```
>>> metre + Time.second
Traceback (...)
...
OperandTypeError: unsupported operand type(s) for +:...
```

However, you can take ratios of incommensurable quantities (quantities with different dimensions), and multiply or divide them.

```
>>> metre/Time.second
>>> 1::{<Length.metre>}/{<Time.second>}
```

Warning: `Decimal numbers` are not supported.

This module makes not attempt to fix the standing incompatibility between floats and `decimal.Decimal`:

```
>>> import decimal
>>> decimal.Decimal('0') + 0.1
Traceback (...)
...
TypeError: unsupported operand type(s) for +: 'Decimal' and 'float'
```

The signature created by *`Dimension`* for its canonical unit is simply a string that varies with the name of the dimension and that of the canonical unit. This implies that you can *recreate* the same dimension and it will be interoperable with the former:

```
>>> @Dimension.new
... class L(object):
...    m = UNIT

>>> m = L.m  # Save this


>>> # Recreate the same dimension.
>>> @Dimension.new
... class L(object):
...    m = UNIT

>>> m == L.m
True
```

Both the dimension name and the canonical unit name *must* be the same for this to work.

Note: We advice to define a dimension only once and import it where needed.

class `xoutil.dim.meta.`**`Dimension`**

A type for quantities.

This is a metaclass for dimensions. Every instance (class) will automatically have the following attributes:

unitname

The name of canonical unit in the dimension. Notice that *aliases* are created after the defined canonical unit. This is the name of the attribute provided in the class definition of the dimension with value equal to *UNIT*.

unit

The canonical *quantity*. This is the quantity 1 (*UNIT*) expressed in terms of the canonical unit.

signature

The canonical *signature* of the quantities.

It's always true that `Quantity(UNIT, self._signature_) == self._unit_`.

The provided dimension *Length* has the canonical quantity *1 metre*:

```
>>> Length.metre
1::{<Length.metre>}/{}

>>> Length._unit_ == Length.metre == Quantity(1, Length._signature_)
True
```

classmethod new(**source*, ***kwargs*)

Define a new dimension.

This is a wrapped decorator. The actual possible signatures are:

- `new(unit_alias=None, unit_aliases=None)(source)`
- `new(source)`

This allows to use this method as decorator with or without arguments.

Parameters

- **source** – A class with at least the canonical unit definition. Other unit definitions will be automatically converted.
- **unit_alias** – An alias for the canonical unit. You cannot use a *source* with several canonical units. This is a simple way to introduce a single alias.
- **unit_aliases** – A sequence with the name of other aliases for the canonical unit.

Example:

```
>>> @Dimension.new(unit_alias='man')
... class Workforce(object):
...     men = UNIT
```

```
>>> Workforce.men == Workforce.man == Workforce._unit_
True
```

The resulting class will be an instance of *Dimension*

```
>>> isinstance(Workforce, Dimension)
True
```

The original class is totally missed:

```
>>> Workforce.mro()
[...Workforce, object]
```

To complete the example, let's introduce the dimension Effort that expresses the usual amount of men-power and time needed to complete some task. However *Time* has the second as it canonical unit, but the standard for Effort is men-hour:

```
>>> class Effort(Workforce * Time):
...     # Since the canonical unit of a composed quantity type is built from
...     # the canonical units of the operands, but the true "canonical type"
...     # of effort is usually men-hour we re-introduce it.
...     men_hour = 60
```

This does not mean that `Effort._unit_ == Effort.men_hour`. The canonical unit would be `Effort.men_second`.

class `xoutil.dim.meta.`**`Signature`**(*top=None*, *bottom=None*)

The layout of the kinds that compose a quantity.

The layout is given by a pair non-ordered collections (repetition is allowed): the numerator (we call it top within the signature) and the denominator (bottom).

We represent a signature as `{top elements}/{bottom elements}`.

You may regard a signature as an abstract 'syntactical' part of a quantity. For Length, the `{metre}/{}` is the signature of such a quantity.

The number "10" is not tied to any particular kind of quantity. Bare numbers have no kind and the bear the signature `{}/{}`.

The items of top and bottom are required to be comparable for equality (==).

You can multiply and divide signatures and simplification happens automatically.

You *should* regard signatures as immutable values. In fact, this is kind of an internal, but interesting, concept of this module.

Examples:

```
>>> distance = Signature('m')
>>> distance
{m}/{}

>>> time = Signature('s')

>>> freq = 1/time
>>> freq
{}/{s}

>>> speed = distance/time
>>> speed
{m}/{s}

>>> acceleration = speed/time
>>> acceleration
{m}/{s, s}
```

You may compare signatures for equality.

```
>>> acceleration == distance/(time*Signature('s'))
True
```

```
>>> speed == distance * freq
True
```

Signature don't support neither addition nor subtraction:

```
>>> distance + distance
Traceback (...)
...
TypeError: unsupported operand type(s) for +: 'Signature' and 'Signature'
```

static **`simplify`**(*top*, *bottom*)

Removes equal items from top and bottom in a one-to-one correspondence.

Signatures are simplified on initialization:

```
>>> Signature('abcxa', 'bxay')
{c, a}/{y}
```

This function takes top and bottom and returns simplified tuples for top and bottom.

class `xoutil.dim.meta.`**`Quantity`**(*quantity*, *units*)

A concrete number of *quantity* (expressed in) *units*.

See also:

https://en.wikipedia.org/wiki/Concrete_number

Parameters

- **`quantity`** – A real number.
- **`units`** – A *`signature`* for the units the denominate the given quantity.

You can construct instances by operating with the attributes of a dimension. For instance, this is 5 kilometres:

```
>>> from xoutil.dim.base import L
>>> 5 * L.km
5000::{<Length.metre>}/{}
```

A concrete number is of the type of its dimension:

```
>>> isinstance(5 * L.km, L)
True
```

class `xoutil.dim.meta.`**`Scalar`**

A quantity whose signature is always *empty*.

Most of the time you should not deal with this quantity. Any normal operation that results in a scalar gets reduced to Python's type:

```
>>> from xoutil.dim.base import L
>>> L.m/L.m
1.0
```

This type makes the operations on *`dimensions`* closed under multiplication:

```
>>> Scalar * L == L == L * Scalar
True
```

`xoutil.dim.meta.`**`UNIT`**

This the constant value `1`. It's given this name to emphasize it's the canonical unit for a dimension.

`xoutil.dim.meta.`**`SCALAR`**

The signature of dimensionless quantities.

`xoutil.dim.base` - The base physical quantities

The standard physical quantities.

class `xoutil.dim.base.`**`Length`**

The Length base quantity.

`metre`

The canonical unit.

`m`

An alias of *`metre`*

Other attributes:

`kilometre`

`km`

`centimetre`

`cm`

`millimetre`

`mm`

`nanometre`

`nm`

class `xoutil.dim.base.`**`Time`**

The Time base quantity.

`second`

The canonical unit.

`s`

An alias of *`second`*

Other attributes:

`millisecond`

`ms`

`nanosecond`

`ns`

`minute`

`hour`

class `xoutil.dim.base.`**`Mass`**

The Mass base quantity.

`kilogram`

The canonical unit.

`kg`

An alias of *`kilogram`*

Other attributes:

`gram`

class `xoutil.dim.base.`**`ElectricCurrent`**
The electrical current base quantity.

`ampere`
The canonical unit.

`A`
An alias of *`ampere`*

class `xoutil.dim.base.`**`Temperature`**
The thermodynamic temperature base quantity.

`kelvin`
The canonical unit.

`K`
An alias of *`kelvin`*

classmethod **`from_celcius`**(*val*)
Convert *val* ºC to K

classmethod **`from_fahrenheit`**(*val*)
Convert *val* ºF to K

class `xoutil.dim.base.`**`Substance`**
The amount of substance.

`mole`

`mol`
An alias of *`mole`*.

class `xoutil.dim.base.`**`Luminosity`**
The luminous intensity base quantity.

`candela`

Aliases

class `xoutil.dim.base.`**`L`**
An alias of *`Length`*

class `xoutil.dim.base.`**`T`**
An alias of *`Time`*

class `xoutil.dim.base.`**`M`**
An alias of *`Mass`*

class `xoutil.dim.base.`**`I`**
An alias of *`ElectricCurrent`*

class `xoutil.dim.base.`**`O`**
An alias of *`Temperature`*. We can’t really use the Greek Theta Θ

class `xoutil.dim.base.`**`N`**
An alias of *`Substance`*

class `xoutil.dim.base.`**`J`**
An alias of *`Luminosity`*

Derived quantities

class `xoutil.dim.base.`**`Area`**

Defined as `L**2`.

`metre_squared`

The canonical unit.

class `xoutil.dim.base.`**`Volume`**

Defined as `L**3`.

`metre_cubic`

The canonical unit.

class `xoutil.dim.base.`**`Frequency`**

Defined as `T**-1` (which is the same as `1/T`).

`unit_per_second`

The canonical unit.

Aliases of the canonical unit:

`Hz`

class `xoutil.dim.base.`**`Force`**

Defined as `L * M * T**-2`.

`metre_kilogram_per_second_squared`

The canonical unit.

Aliases of the canonical unit:

`N`

`Newton`

class `xoutil.dim.base.`**`Presure`**

Defined as `L**-1 * M * T**-2`.

`kilogram_per_metre_per_second_squared`

Aliases of the canonical unit:

`pascal`

`Pa`

class `xoutil.dim.base.`**`Velocity`**

Defined as `L * T**-1`.

`metre_per_second`

The canonical unit.

class `xoutil.dim.base.`**`Acceleration`**

Defined as `L * T**-2`.

`metre_per_second_squared`

The canonical unit.

On the automatically created names for derived quantities

We automatically create the name of the canonical unit of quantities derived from others by simple rules:

- `A * B` joins the canonical unit names together with a low dash '_' in-between. Let's represent it as *a_b*, where *a* stands for the name of the canonical unit of `A` and *b*, the canonical unit of `B`.

  For the case, `A * A` the unit name is *a_squared*.
- `A/B` gets the name *a_per_b*. `1/A` gets the name *unit_per_a*
- `A**n`; when `n=1` this is the same as `A`; when `n=2` this is the same as `A * A`; for other positive values of `n`, the canonical unit name is *a_pow_n*; for negative values of `n` is the same as `1/A**n`; for `n=0` this is the *`Scalar`* quantity.

`xoutil.dim.currencies` – Concrete numbers for money

Concrete numbers for money.

You may have 10 dollars and 5 euros in your wallet, that does not mean that you have 15 of anything (but bills, perhaps). Though you may *evaluate* your cash in any other currency you don't have that value until you perform an exchange with a given rate.

This module support the family of currencies. Usage:

```
>>> from xoutil.dim.currencies import Rate, Valuation, currency
>>> dollar = USD = currency('USD')
>>> euro = EUR = currency('EUR')
>>> rate = 1.19196 * USD/EUR

>>> isinstance(dollar, Valuation)
True

>>> isinstance(rate, Rate)
True

# Even 0 dollars are a valuation
>>> isinstance(dollar - dollar, Valuation)
True

# But 1 is not a value nor a rate
>>> isinstance(dollar/dollar, Valuation) or isinstance(dollar/dollar, Rate)
False
```

Currency names are case-insensitive. We don't check the currency name is listed in ISO 4217. So currency `MVA` is totally acceptable in this module.

We don't download rates from any source.

This module allows you to trust your computations of money by allowing only sensible operations:

```
>>> dollar + euro
Traceback (...)
...
OperandTypeError: unsupported operand type(s) for +: '{USD}/{}' and '{EUR}/{}
```

If you convert your euros to dollars:

```
>>> dollar + rate * euro
2.19196::{USD}/{}

# Or your dollars to euros
```

```
>>> dollar/rate + euro
1.83895432733::{EUR}/{}
```

`xoutil.eight` – Extensions for writing code that runs on Python 2 and 3

Todo

check automodule:: xoutil.eight :members:

The name comes from (Manu's idea') "2 raised to the power of 3".

This module is divided in several parts.

`xoutil.eight.abc` - Abstract Base Classes (ABCs) according to PEP 3119

Abstract Base Classes (ABCs) according to PEP 3119.

Compatibility module between Python 2 and 3.

This module defines one symbol that is defined in Python 3 as a class:

> **class ABC(metaclass=ABCMeta):** """"Helper class that provides a standard way to create an ABC using inheritance. """ pass

In our case it's defined as `ABC = metaclass(ABCMeta)`, that is a little tricky (see *xoutil.eight.meta.metaclass*:func').

abstractclassmethod is deprecated. Use *classmethod* with *abstractmethod* instead.

abstractstaticmethod is deprecated. Use *staticmethod* with *abstractmethod* instead.

`xoutil.eight.abc.`**`get_cache_token`**`()`

Returns the current ABC cache token.

The token is an opaque object (supporting equality testing) identifying the current version of the ABC cache for virtual sub-classes. The token changes with every call to `register()` on any ABC.

`xoutil.eight.meta` - *metaclass* function using Python 3 syntax

Implements the metaclass() function using the Py3k syntax.

`xoutil.eight.meta.`**`metaclass`**`(`*meta*, ***kwargs*`)`

Define the metaclass of a class.

New in version 1.7.0: It's available as `xoutil.objects.metaclass()` since 1.4.1. That alias is now deprecated and will be removed in 1.8.

This function allows to define the metaclass of a class equally in Python 2 and 3.

Usage:

```
>>> class Meta(type):
...   pass

>>> class Foobar(metaclass(Meta)):
...   pass

>>> class Spam(metaclass(Meta), dict):
...   pass

>>> type(Spam) is Meta
True

>>> Spam.__bases__ == (dict, )
True
```

New in version 1.5.5: The *kwargs* keywords arguments with support for `__prepare__`.

Metaclasses are allowed to have a `__prepare__` classmethod to return the namespace into which the body of the class should be evaluated. See **PEP 3115**.

Warning: The **PEP 3115** is not possible to implement in Python 2.7.

Despite our best efforts to have a truly compatible way of creating meta classes in both Python 2.7 and 3.0+, there is an inescapable difference in Python 2.7. The **PEP 3115** states that `__prepare__` should be called before evaluating the body of the class. This is not possible in Python 2.7, since `__new__` already receives the attributes collected in the body of the class. So it's always too late to call `__prepare__` at this point and the Python 2.7 interpreter does not call it.

Our approach for Python 2.7 is calling it inside the `__new__` of a "side" metaclass that is used for the base class returned. This means that `__prepare__` is called **only** for classes that use the `metaclass()` directly. In the following hierarchy:

```
class Meta(type):
    @classmethod
    def __prepare__(cls, name, bases, **kwargs):
        from xoutil.future.collections import OrderedDict
        return OrderedDict()

class Foo(metaclass(Meta)):
    pass

class Bar(Foo):
    pass
```

when creating the class `Bar` the `__prepare__()` class method is not called in Python 2.7!

See also:

`xoutil.future.types.prepare_class()` and `xoutil.future.types.new_class()`.

Warning: You should always place your metaclass declaration *first* in the list of bases. Doing otherwise triggers *twice* the metaclass' constructors in Python 3.1 or less.

If your metaclass has some non-idempotent side-effect (such as registration of classes), then this would lead to unwanted double registration of the class:

```
>>> class BaseMeta(type):
...     classes = []
...     def __new__(cls, name, bases, attrs):
...         res = super(BaseMeta, cls).__new__(cls, name, bases, attrs)
...         cls.classes.append(res)    # <-- side effect
...         return res

>>> class Base(metaclass(BaseMeta)):
...     pass

>>> class SubType(BaseMeta):
...     pass

>>> class Egg(metaclass(SubType), Base):   # <-- metaclass first
...     pass

>>> Egg.__base__ is Base   # <-- but the base is Base
True

>>> len(BaseMeta.classes) == 2
True

>>> class Spam(Base, metaclass(SubType)):
...     'Like "Egg" but it will be registered twice in Python 2.x.'
```

In this case the registration of Spam ocurred twice:

```
>>> BaseMeta.classes
[<class Base>, <class Egg>, <class Spam>, <class Spam>]
```

Bases, however, are just fine:

```
>>> Spam.__bases__ == (Base, )
True
```

New in version 1.7.1: Now are accepted atypical meta-classes, for example functions or any callable with the same arguments as those that type accepts (class name, tuple of base classes, attributes mapping).

xoutil.eight.mixins - functions to create helper classes and mixins

Two functions to create helper classes and mixins.

This module is in the *eight* context because these two functions depend on several concepts that are different in Python 2 and 3.

- *helper_class()* creates a base class that represent a meta-class. For example (only for Python 3), *xoutil.eight.abc.ABC* is different to *abc.ABC*:

```
>>> from xoutil.eight.abc import ABC, ABCMeta
>>> class One(ABC):
...     pass
>>> One.__bases__ == (ABC, )
False
>>> One.__bases__ == (Mixin, )
True
```

```
>>> from abc import ABC
>>> class Two(ABC):
...     pass
>>> Two.__bases__ == (ABC, )
True
>>> Two.__bases__ == (Mixin, )
False
```

- *mixin()* create a base-class tha consolidate several mix-ins and meta-classes. For example:

```
>>> from xoutil.eight.abc import ABCMeta

>>> class One(dict):
...     pass

>>> class Two(object):
...     pass

>>> class OneMeta(type):
...     pass

>>> class TwoMeta(type):
...     pass

>>> Test = mixin(One, Two, meta=[OneMeta, TwoMeta, ABCMeta], name='Test')
>>> Test.__name__ == 'Test'
True
>>> isinstance(Test, ABCMeta)
True
```

These modules (this one and *meta*) must have four utilities:

- *metaclass* to use a unique syntax to declare meta-classes between Python 2 and 3.
- *helper_class* to build a class that when used as a base impose a meta-class and not is found in resulting bases of defined class. For example *xoutil.eight.abc.ABC*.
- *mixin* build a mixin-base composing several parts and meta-classes.
- *compose* specify the use of a mixin as one of the bases, but the new defined class will not be a mixin, this is not implemented yet because will require a big architecture re-factoring; for example:

```
class Foobar(MyBase, compose(MyMixin)):
    pass
```

Maybe the last two names must be interchanged.

`xoutil.eight.mixins.`**`helper_class`**`(`*meta*, *name=None*`)`

Create a helper class based in the meta-class concept.

Parameters

- **meta** – The meta-class type to base returned helper-class on it.
- **name** – The name (`__name__`) to assign to the returned class; if None is given, a nice name is calculated.

For example:

```
>>> from abc import ABCMeta
>>> ABC = helper_class(ABCMeta)     # better than Python 3's abc.ABC :(
>>> class MyError(Exception, ABC):
...     pass
>>> (MyError.__bases__ == (Exception,), hasattr(MyError, 'register'))
(True, True)
```

This function calls `metaclass()` internally. So, in the example anterior, *MyError* declaration is equivalent to:

```
>>> class MyError(Exception, metaclass(ABCMeta)):
...     pass
```

`xoutil.eight.mixins.`**`mixin`**`(`*`*args`*`, `*`**kwargs`*`)`

Weave a *mixin*.

Parameters of this function are a little tricky.

Parameters

- **`name`** – The name of the new class created by this function. Could be passed as positional or keyword argument. Use `__name__` as an alias. See *`helper_class()`* for more info about this parameter and next two.
- **`doc`** – Documentation of the returned mixin-class. Could be passed as positional or keyword argument. Use `__doc__` as an alias.
- **`module`** – Always given as a keyword parameter. A string -or an object to be used as reference- representing the module name. Use `__module__` as an alias.
- **`metaclass`** – Always given as a keyword parameter. Could be one type value or a list of values (multiples meta-classes). Use (`__metaclass__`, `metaclasses`, or `meta`) as aliases.

If several mixins with the same base are used all-together in a class inheritance, Python generates `TypeError: multiple bases have instance lay-out conflict`. To avoid that, inherit from the class this function returns instead of desired *base*.

`xoutil.eight.string` - Checkers for simple types

Technical string handling.

Technical strings are those that requires to be instances of *str* standard type. See *String Ambiguity in Python* for more information.

This module will be used mostly as a namespace, for example:

```
from xoutil.eight import string
Foobar.__name__ = string.force(class_name)
```

If these functions are going to be used standalone, do something like:

```
from xoutil.eight.string import force as force_str
Foobar.__name__ = force_str(class_name)
```

`xoutil.eight.string.`**`check_identifier`**`(`*`s`*`)`

Check if *s* is a valid identifier.

`xoutil.eight.string.`**`force`**`(`*`value=''`*`)`

Convert any value to standard *str* type in a safe way.

This function is useful in some scenarios that require *str* type (for example attribute `__name__` in functions and types).

As `str is bytes` in Python 2, using str(value) assures correct these scenarios in most cases, but in other is not enough, for example:

```
>>> from xoutil.eight import string
>>> def inverted_partial(func, *args, **keywords):
...     def inner(*a, **kw):
...         a += args
...         kw.update(keywords)
...         return func(*a, **kw)
...     name = func.__name__.replace('lambda', u'λ')
...     inner.__name__ = string.force(name)
...     return inner
```

`xoutil.eight.string.`**`force_ascii`**(*value*)
Return the string normal form for the *value*

Convert all non-ascii to valid characters using unicode 'NFKC' normalization.

`xoutil.eight.string.`**`isfullidentifier`**(*s*)
Check if *arg* is a valid dotted Python identifier.

See *`isidentifier()`* for what "validity" means.

`xoutil.eight.string.`**`isidentifier`**(*s*)
If *s* is a valid identifier according to the language definition.

`xoutil.eight.string.`**`safe_isfullidentifier`**(*s*)
Check if *arg* is a valid dotted Python identifier.

Check before if *s* is instance of string types. See *`safe_isidentifier()`* for what "validity" means.

`xoutil.eight.string.`**`safe_isidentifier`**(*s*)
If *s* is a valid identifier according to the language definition.

Check before if *s* is instance of string types.

`xoutil.eight.string.`**`safe_join`**(*separator*, *iterable*)
Similar to *join* method in string objects.

The semantics is equivalent to `separator.join(iterable)` but forcing separator and items to be of `str` standard type.

For example:

```
>>> safe_join('-', range(6))
'0-1-2-3-4-5'
```

Check that the expression `'-'.join(range(6))` raises a `TypeError`.

`xoutil.eight.text` - TODO

Text handling, strings can be part of internationalization processes.

See *String Ambiguity in Python* for more information.

New in version 1.8.0.

`xoutil.eight.text.`**`force`**`(`*buffer*, *encoding=None*`)`
Convert any value to standard *text* type in a safe way.

The standard text type is `unicode` in Python 2 and `str` in Python 3.

`xoutil.eight.text.`**`safe_join`**`(`*separator*, *iterable*, *encoding=None*`)`
Similar to *join* method in string objects.

The semantics is equivalent to `separator.join(iterable)` but forcing separator and items to be the valid instances of standard *text* type (`unicode` in Python 2 and `str` in Python 3).

For example:

```
>>> safe_join('-', range(6))
'0-1-2-3-4-5'
```

Check that the expression `'-'.join(range(6))` raises a `TypeError`.

Parameters **`encoding`** – used to allow control, but won't be common to use it.

`xoutil.eight.io` - Extensions to Python's *io* module

Extensions to Python's `io` module.

You may use it as drop-in replacement of `io`. Although we don't document all items here. Refer to `io` documentation.

In Python 2, buil-int `open()` is different from `io.open()`; in Python 3 are the same function.

So, generated files with the built-in funtion in Python 2, can not be processed using *abc* types, for example:

```
f = open('test.rst')
assert isinstance(f, io.IOBase)
```

will fail in Python 2 and not in Python 3.

Another incompatibilities:

- *file* type doesn't exists in Python 3.
- Python 2 instances created with *io.StringIO*:class', or with `io.open()` using text mode, don't accept *str* values, so it will be better to use any of the standards classes (`StringIO.StringIO`, `cStringIO.StringIO` or `open()` built-in).

New in version 1.7.0.

`xoutil.eight.io.`**`is_file_like`**`(`*obj*`)`
Return if *obj* is a valid file type or not.

`xoutil.eight.queue` - A multi-producer, multi-consumer queue

A multi-producer, multi-consumer queue.

`xoutil.eight.exceptions` - Exceptions handling compatibility

Solve compatibility issues for exceptions handling.

Python 2 defines a module named *exceptions* but Python 3 doesn't. We decided not to implement something similar, for example, in *`xoutil.future`* package because all these exception classes are built-ins in both Python major versions, so use any of them directly; nevertheless `StandardError` is undefined in Python 3, we introduce some adjustments here in base classes (`BaseException` and `StandardError` classes).

The functions *`catch()`* and *`throw()`* unify syntax differences raising exceptions. In Python 2 the syntax for `raise` is:

```
"raise" [type ["," value ["," traceback]]]
```

and in Python 3:

```
"raise" [error[.with_traceback(traceback)] ["from" cause]]
```

You can use *`catch()`* as a function to wrap errors going to be raised with a homogeneous syntax using a *trace* extra argument:

```
>>> divisor = 0
>>> try:
...     inverted = 1/divisor
... except BaseException:
...     raise catch(ValueError('Invalid divisor.'))
```

If you want to be completely compatible raising exceptions with trace-backs, use the *`throw()`* function instead the `raise` statement.

`xoutil.eight.exceptions.`**`catch`**`(`*self*`)`
Check an error to settle trace-back information if found.

Parameters **`self`** – The exception to check.

`xoutil.eight.exceptions.`**`caught`** **= <xoutil.tasking.AutoLocal object>**
Last caught trace context, see *`catch()`*.

`xoutil.eight.exceptions.`**`grab`**`(`*self=None*, *trace=None*`)`
Prepare an error being raised with a trace-back and/or a cause.

Parameters

- **`self`** – The exception to be raised or None to capture the current trace context for future use.
- **`trace`** – Could be a trace-back, a cause (exception instance), or both in a tuple (or list) with `(cause, traceback)`. If None, use the current system exception info as the trace (see `sys.exc_info()` built-in function).

This function create a syntax for `raise` statement, compatible for both major Python versions.

`xoutil.eight.exceptions.`**`throw`**`(`*error*, *tb=None*`)`
Unify syntax for raising an error with trace-back information.

Instead of using the Python `raise` statement, use `throw(error, tb)`. If *tb* argument is not given, the trace-back information is looked up in the context.

`xoutil.eight.exceptions.`**`traceof`**`(`*error*`)`
Get the trace-back information of the given *error*.

Return None if not defined.

`xoutil.eight.exceptions.`**`with_cause`**`(`*self*, *cause*`)`
set self.__cause__ to *cause* and return self.

`xoutil.eight.exceptions.`**`with_traceback`**`(`*self*, *tb*`)`
set self.__traceback__ to *tb* and return self.

`xoutil.formatter` - Formatting

Smart formatting.

class `xoutil.formatter.`**`Template`**`(`*template*`)`

A string class for supporting $-substitutions.

It has similar interface that *string.Template* but using "eval" instead simple dictionary looking.

This means that you get all the functionality provided by *string.Template* (although, perhaps modified) and you get also the ability to write more complex expressions.

If you need repetition or other flow-control sentences you should use other templating system.

If you enclose and expression within `${?...}` it will be evaluated as a python expression. Simple variables are allowed just with `$var` or `${var}`:

```
>>> tpl = Template(str('${?1 + 1} is 2, and ${?x + x} is $x + ${x}'))
>>> (tpl % dict(x=4)) == '2 is 2, and 8 is 4 + 4'
True
```

The mapping may be given by calling the template:

```
>>> tpl(x=5) == '2 is 2, and 10 is 5 + 5'
True
```

`xoutil.formatter.`**`count`**`(`*source*, *chars*`)`

Counts how chars from *chars* are found in *source*:

```
>>> count('Todos los nenes del mundo vamos una rueda a hacer', 'a')
1

# The vowel "i" is missing
>>> count('Todos los nenes del mundo vamos una rueda a hacer', 'aeiuo')
4
```

`xoutil.fp` – Functional Programming in Python

Advanced functional programming in Python.

Note: This module is in **EXPERIMENTAL** state, we encourage not to use it before declared stable.

Ideally, a function only takes inputs and produce outputs, and doesn't have any internal state that affects the output produced for a given input (like in Haskell).

Contents

`xoutil.fp.option` - Functional Programming Option Type

Functional Programming *Option Type* definition.

In Programming, and Type Theory, an *option type*, or *maybe type*, represents encapsulation of an optional value; e.g., it is used in functions which may or may not return a meaningful value when they are applied.

It consists of either a constructor encapsulating the original value `x` (written `Just x` or `Some x`) or an empty constructor (called *None* or *Nothing*). Outside of functional programming, these are known as *nullable types*.

In our case *option type* will be the *`Maybe`* class (the equivalent of *Option* in *Scala Programming Language*), the wrapper for valid values will be the *`Just`* class (equivalent of *Some* in *Scala*); and the wrapper for invalid values will be the *`Wrong`* class.

Instead of *None* or *Nothing*, *Wrong* is used because two reasons: (1) already existence of *None* special Python value, and (2) *`Wrong`* also wraps incorrect values and can have several instances (not only a *null* value).

class `xoutil.fp.option.`**`Just`**`(`**args*`)`

A wrapper for valid results.

class `xoutil.fp.option.`**`Maybe`**`(`**args*`)`

Wrapper for optional values.

The Maybe type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented as `Just a`), or it is empty (represented as `Nothing`). Using *Maybe*‘ is a good way to deal with errors or exceptional cases without resorting to drastic measures such as error. In this implementation we make a variation where a `Wrong` object represents a missing (with special value `Nothing`) or an improper value (including errors).

See descendant classes *`Just`* and *`Wrong`* for more information.

This implementation combines `Maybe` and `Either` Haskell data types. `Maybe` is a means of being explicit that you are not sure that a function will be successful when it is executed. Conventionally, the usage of `Either` for errors uses `Right` when the computation is successful, and `Left` for failing scenarios.

In this implementation, *Just*:class‘ us used for equivalence with both Haskell `Just` and `Right` types; *`Wrong`* is used with the special value `Nothing` and to encapsulate errors or incorrect values (Haskell `Left`).

Haskell:

```
data Maybe a = Nothing | Just a

either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Case analysis for the Either type. If the value is Left a, apply the first function to a; if it is Right b, apply the second function to b.

classmethod **`choose`**`(`**types*`)`

Decorator to force *Maybe* values constraining to expecting types.

For example, a function that return a collection (tuple or list) if valid or False if not, if not decorated could be ambiguous for an empty collection:

```
>>> @Just.choose(tuple, list)
... def check_range(values, min, max):
...     if isinstance(values, (tuple, list)):
...         return [v for v in values if min <= v <= max]
...     else:
...         return False

>>> check_range(range(10), 7, 17)
[7, 8, 9]

>>> check_range(range(10), 17, 27)
Just([])
```

```
>>> check_range(set(range(10)), 7, 17)
False
```

classmethod compel(*value*)

Coerce to the correspondent logical Boolean value.

Just is logically true, and *Wrong* is false.

For example:

```
>>> Just.compel([1])
[1]

>>> Just.compel([])
Just([])

>>> Wrong.compel([1])
Wrong([1])

>>> Wrong.compel([])
[]
```

classmethod triumph(*value*)

Coerce to a logical Boolean value.

A wrapper *Just* is logically true, and *Wrong* is false.

For example:

```
>>> Just.triumph([1])
[1]

>>> Just.triumph([])
Just([])

>>> Wrong.triumph([1])
Wrong([1])

>>> Wrong.triumph([])
[]
```

class xoutil.fp.option.**Wrong**(**args*)

A wrapper for invalid results.

When encapsulation errors, the current trace-back is properly encapsulated using *xoutil.eight.exceptions* module features.

# TODO: Use *naught* or *Left* instead.

xoutil.fp.option.**false = Wrong(False)**

A *Wrong* special singleton encapsulating the *False* value.

xoutil.fp.option.**none = Wrong(None)**

A *Wrong* special singleton encapsulating the *None* value.

xoutil.fp.option.**take**(*value*)

Extract a value.

xoutil.fp.option.**true = Just(True)**

A *Just* special singleton encapsulating the *True* value.

Further Notes

It could be thought that this kind of concept is useless in Python because the dynamic nature of the language, but always there are certain logic systems that need to wrap "correct" false values and "incorrect" true values.

Also, in functional programming, errors can be reasoned in a new way: more like as *error values* than in *exception handling*. Where the `Maybe` type expresses the failure possibility through `Wrong` instances encapsulating errors.

When receiving a `Wrong` instance encapsulating an error, and want to recover the *exception propagation style* - instead of continue in *pure functional programming*-, to re-raise the exception, instead the *raise* Python statement, use `throw()`.

See https://en.wikipedia.org/wiki/Monad_%28functional_programming%29#The_Maybe_monad

`xoutil.fp.prove` - Prove validity of values

Proving success or failure of a function call has two main patterns:

1. Predicative: a function call returns one or more values indicating a failure, for example method `find` in strings returns `-1` if the sub-string is not found. In general this pattern considers a set of values as logical Boolean true, an other set false.

   Example:

   ```
   index = s.find('x')
   if index >= 0:
       ...     # condition of success
   else:
       ...     # condition of failure
   ```

2. Disruptive: a function call throws an exception on a failure breaking the normal flow of execution, for example method `index` in strings.

   Example:

   ```
   try:
       index = s.index('x)
   except ValueError:
       ...     # condition of failure
   else:
       ...     # condition of success
   ```

   The exception object contains the semantics of the ""anomalous condition". Exception handling can be used as flow control structures for execution context inter-layer processing, or as a termination condition.

Module content

Validity proofs for data values.

There are some basic helper functions:

- `predicative()` wraps a function in a way that a logical false value is returned on failure. If an exception is raised, it is returned wrapped as an special false value. See `Maybe` monad for more information.
- `vouch()` wraps a function in a way that an exception is raised if an invalid value (logical false by default) is returned. This is useful to call functions that use "special" false values to signal a failure.

- `enfold()` creates a decorator to convert a function to use either the `predicative()` or the `vouch()` protocol.

New in version 1.8.0.

`xoutil.fp.prove.`**`enfold`**`(`*checker*`)`

Create a decorator to execute a function inner a safety wrapper.

Parameters **`checker`** – Could be any function to enfold, but it's intended mainly for `predicative()` or `vouch()` functions.

In the following example, the semantics of this function can be seen. The definition:

```
>>> @enfold(predicative)
... def test(x):
...     return 1 <= x <= 10

>>> test(5)
5
```

It is equivalent to:

```
>>> def test(x):
...     return 1 <= x <= 10

>>> predicative(test, 5)
5
```

In other hand:

```
>>> @enfold(predicative)
... def test(x):
...     return 1 <= x <= 10

>>> test(15)
5
```

`xoutil.fp.prove.`**`predicative`**`(`*function*, **args*, ***kwds*`)`

Call a function in a safety wrapper returning a false value if fail.

This converts any function into a predicate. A predicate can be thought as an operator or function that returns a value that is either true or false.

Predicates are sometimes used to indicate set membership: on certain occasions it is inconvenient or impossible to describe a set by listing all of its elements. Thus, a predicate `P(x)` will be true or false, depending on whether x belongs to a set.

If the argument *function* validates its arguments, return a valid true value. There are two special conditions: first, a value treated as false for Python conventions (for example, `0`, or an empty string); and second, when an exception is raised; in both cases the predicate will return an instance of `Maybe`.

`xoutil.fp.prove.`**`vouch`**`(`*function*, **args*, ***kwds*`)`

Call a function in a safety wrapper raising an exception if it fails.

When the wrapped function fails, an exception must be raised. A predicate fails when it returns a false value. To avoid treat false values of some types as fails, use `Just` to return that values wrapped.

`xoutil.fp.tools` – High-level pure function tools

Tools for working with functions in a more "pure" way.

class `xoutil.fp.tools.`**`compose`**`(`**funcs*`)`

Composition of several functions.

Functions are composed right to left. A composition of zero functions gives back the *`identity()`* function.

Rules must be fulfilled (those inner *all*):

```
>>> x = 15
>>> f, g, h = x.__add__, x.__mul__, x.__xor__
>>> all((compose() is identity,
...
...      # identity functions are optimized
...      compose(identity, f, identity) is f,
...
...      compose(f) is f,
...      compose(g, f)(x) == g(f(x)),
...      compose(h, g, f)(x) == h(g(f(x)))))
True
```

If any "intermediate" function returns an instance of:

- *`pos_args`*: it's expanded as variable positional arguments to the next function.
- *`kw_args`*: it's expanded as variable keyword arguments to the next function.
- *`full_args`*: it's expanded as variable positional and keyword arguments to the next function.

The expected usage of these is **not** to have function return those types directly, but to use them when composing functions that return tuples and expect tuples.

`xoutil.fp.tools.`**`identity`**`(`*arg*`)`

Returns its argument unaltered.

class `xoutil.fp.tools.`**`pos_args`**

Mark variable number positional arguments (see `fargs`).

class `xoutil.fp.tools.`**`kw_args`**

Mark variable number keyword arguments (see `fargs`).

class `xoutil.fp.tools.`**`full_args`**

Mark variable number arguments for composition.

Pair containing positional and keyword `(args, kwds)` arguments.

In standard functional composition, the result of a function is considered a single value to be use as the next function argument. You can override this behaviour returning one instance of *`pos_args`*, *`kw_args`*, or this class; in order to provide multiple arguments to the next call.

Since types are callable, you may use it directly in *`compose()`* instead of changing your functions to returns the instance of one of these classes:

```
>>> def join_args(*args):
...     return ' -- '.join(str(arg) for arg in args)

>>> compose(join_args, pos_args, list, range)(2)
'0 -- 1'

# Without 'pos_args', it prints the list
>>> compose(join_args, list, range)(2)
'[0, 1]'
```

`xoutil.fs` – file system utilities

File system utilities.

This module contains file-system utilities that could have side-effects. For path-handling functions that have no side-effects look at `xoutil.fs.path`.

`xoutil.fs.`**`ensure_filename`**(*filename*, *yields=False*)

Ensures the existence of a file with a given filename.

If the filename is taken and is not pointing to a file (or a link to a file) an OSError is raised. If *exist_ok* is False the filename must not be taken; an OSError is raised otherwise.

The function creates all directories if needed. See `makedirs()` for restrictions.

If *yields* is True, returns the file object. This way you may open a file for writing like this:

```
with ensure_filename('/tmp/good-name-87.txt', yields=True) as fh:
    fh.write('Do it!')
```

The file is open in mode 'w+b'.

New in version 1.6.1: Added parameter *yield*.

`xoutil.fs.`**`imap`**(*func*, *pattern*)

Yields *func(file_0, stat_0)*, *func(file_1, stat_1)*, ... for each dir path. The *pattern* may contain:

- Simple shell-style wild-cards à la *fnmatch*.
- Regex if pattern starts with '(?'. Expressions must be valid, as in "(?:[^.].*)$" or "(?i).*.jpe?g$". Remember to add the end mark '$' if needed.

`xoutil.fs.`**`iter_dirs`**(*top='.'*, *pattern=None*, *regex_pattern=None*, *shell_pattern=None*)

Iterate directories recursively.

The params have analagous meaning that in `iter_files()` and the same restrictions.

`xoutil.fs.`**`iter_files`**(*top='.'*, *pattern=None*, *regex_pattern=None*, *shell_pattern=None*, *followlinks=False*, *maxdepth=None*)

Iterate filenames recursively.

Parameters

- **`top`** – The top directory for recurse into.
- **`pattern`** – A pattern of the files you want to get from the iterator. It should be a string. If it starts with "(?" it will be regarded as a regular expression, otherwise a shell pattern.
- **`regex_pattern`** – An *alternative* to *pattern*. This will always be regarded as a regular expression.
- **`shell_pattern`** – An *alternative* to *pattern*. This should be a shell pattern.
- **`followlinks`** – The same meaning that in *os.walk*.

  New in version 1.2.1.
- **`maxdepth`** – Only files above this level will be yielded. If None, no limit is placed.

  New in version 1.2.1.

Warning: It's an error to pass more than pattern argument.

`xoutil.fs.`**`listdir`**(*path*)

Same as `os.listdir` but normalizes *path* and raises no error.

`xoutil.fs.`**`rmdirs`**(*top='.'*, *pattern=None*, *regex_pattern=None*, *shell_pattern=None*, *exclude=None*, *confirm=None*)

Removes all empty dirs at *top*.

Parameters

- **`top`** – The top directory to recurse into.
- **`pattern`** – A pattern of the dirs you want to remove. It should be a string. If it starts with "(?" it will be regarded as a regular expression, otherwise a shell pattern.
- **`exclude`** – A pattern of the dirs you DON'T want to remove. It should be a string. If it starts with "(?" it will be regarded as a regular expression, otherwise a shell pattern. This is a simple commodity to have you not to negate complex patterns.
- **`regex_pattern`** – An *alternative* to *pattern*. This will always be regarded as a regular expression.
- **`shell_pattern`** – An *alternative* to *pattern*. This should be a shell pattern.
- **`confirm`** – A callable that accepts a single argument, which is the path of the directory to be deleted. *confirm* should return True to allow the directory to be deleted. If *confirm* is None, then all matched dirs are deleted.

Note: In order to avoid common mistakes we won't attempt to remove mount points.

New in version 1.1.3.

`xoutil.fs.`**`stat`**(*path*)

Return file or file system status.

This is the same as the function `os.stat` but raises no error.

`xoutil.fs.`**`walk_up`**(*start*, *sentinel*)

Given a *start* directory walk-up the file system tree until either the FS root is reached or the *sentinel* is found.

The *sentinel* must be a string containing the file name to be found.

Warning: If *sentinel* is an absolute path that exists this will return *start*, no matter what *start* is (in windows this could be even different drives).

If *start* path exists but is not a directory an OSError is raised.

`xoutil.fs.`**`concatfiles`**(**files*, *target*)

Concat several files to a single one.

Each positional argument must be either:

- a file-like object (ready to be passed to `shutil.copyfileobj()`)
- a string, the file path.

The last positional argument is the target. If it's file-like object it must be open for writing, and the caller is the responsible for closing it.

Alternatively if there are only two positional arguments and the first is a collection, the sources will be the members of the first argument.

`xoutil.fs.`**`makedirs`**(*path*, *mode=0o777*, *exist_ok=False*)

Recursive directory creation function. Like `os.mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The default *mode* is `0o777` (octal). On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out.

If *exist_ok* is `False` (the default), an `OSError` is raised if the target directory already exists.

Note: *`makedirs()`* will become confused if the path elements to create include `os.pardir` (eg. ".." on UNIX systems).

This function handles UNC paths correctly.

Changed in version 1.6.1: Behaves as Python 3.4.1.

Before Python 3.4.1 (ie. xoutil 1.6.1), if *exist_ok* was `True` and the directory existed, *`makedirs()`* would still raise an error if *mode* did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See the original `os.makedirs()`.

Contents:

`xoutil.fs.path` – Path utilities

Extensions to os.path

Functions inside this module must not have side-effects on the file-system. This module re-exports (without change) several functions from the `os.path` standard module.

`xoutil.fs.path.`**`join`**(*base*, **extras*)

Join two or more pathname components, inserting '/' as needed.

If any component is an absolute path, all previous path components will be discarded.

Normalize path (after join parts), eliminating double slashes, etc.

`xoutil.fs.path.`**`fix_encoding`**(*name*, *encoding=None*)

Fix encoding of a file system resource name.

encoding is ignored if *name* is already a *str*.

`xoutil.fs.path.`**`normalize_path`**(*base*, **extras*)

Normalize path by:

- expanding '~' and '~user' constructions.
- eliminating double slashes
- converting to absolute.

`xoutil.fs.path.`**`shorten_module_filename`**(*filename*)

A filename, normally a module o package name, is shortened looking his head in all python path.

`xoutil.fs.path.`**`shorten_user`**(*filename*)

A filename is shortened looking for the (expantion) $HOME in his head and replacing it by '~'.

`xoutil.fs.path.`**`rtrim`**(*path*, *n=1*)

Trims the last *n* components of the pathname *path*.

This basically applies *n* times the function *os.path.dirname* to *path*.

path is normalized before proceeding (but not tested to exists).

Changed in version 1.5.5: *n* defaults to 1. In this case rtrim is identical to `os.path.dirname()`.

Example:

```
>>> rtrim('/tmp/a/b/c/d', 3)
'/tmp/a'

# It does not matter if `/` is at the end
>>> rtrim('/tmp/a/b/c/d/', 3)
'/tmp/a'
```

`xoutil.future` - Extend standard modules with "future" features

Extend standard modules including "future" features in current versions.

Version 3 introduce several concepts in standard modules. Sometimes these features are implemented in the evolution of 2.7.x versions. By using sub-modules, these differences can be avoided transparently. For example, you can import `xoutil.future.collections.UserDict` in any version, that it's equivalent to Python 3 `collections.UserDict`, but it don't exists in Python 2.

New in version 1.7.2.

Contents

`xoutil.future.codecs` - Codec registry, base classes and tools

This module extends the standard library's `functools`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

`xoutil.future.codecs.`**`force_encoding`**(*encoding=None*)

Validates an encoding value; if None use *locale.getlocale()[1]*; else return the same value.

New in version 1.2.0.

Changed in version 1.8.0: migrated to 'future.codecs'

`xoutil.future.codecs.`**`safe_decode`**(*s*, *encoding=None*)

Similar to bytes *decode* method returning unicode.

Decodes *s* using the given *encoding*, or determining one from the system.

Returning type depend on python version; if 2.x is *unicode* if 3.x *str*.

New in version 1.1.3.

Changed in version 1.8.0: migrated to 'future.codecs'

`xoutil.future.codecs.`**`safe_encode`**(*u*, *encoding=None*)

Similar to unicode *encode* method returning bytes.

Encodes *u* using the given *encoding*, or determining one from the system.

Returning type is always *bytes*; but in python 2.x is also *str*.

New in version 1.1.3.

Changed in version 1.8.0: migrated to 'future.codecs'

`xoutil.future.collections` - High-performance container datatypes

This module extends the standard library's `collections`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since this is different in Python 2.7 and Python 3.3. Notably importing `abc` is not available in Python 2.7.

We have backported several Python 3.3 features but not all.

class `xoutil.future.collections.`**`defaultdict`**

A hack for `collections.defaultdict` that passes the key and a copy of self as a plain dict (to avoid infinity recursion) to the callable.

Examples:

```
>>> from xoutil.future.collections import defaultdict
>>> d = defaultdict(lambda key, d: 'a')
>>> d['abc']
'a'
```

Since the second parameter is actually a dict-copy, you may (naively) do the following:

```
>>> d = defaultdict(lambda k, d: d[k])
>>> d['abc']
Traceback (most recent call last):
    ...
KeyError: 'abc'
```

You may use this class as a drop-in replacement for `collections.defaultdict`:

```
>>> d = defaultdict(lambda: 1)
>>> d['abc']
1
```

class `xoutil.future.collections.`**`opendict`**

A dictionary implementation that mirrors its keys as attributes:

```
>>> d = opendict({'es': 'spanish'})
>>> d.es
'spanish'

>>> d['es'] = 'espanol'
>>> d.es
'espanol'
```

Setting attributes *does not* makes them keys.

class `xoutil.future.collections.`**`Counter`**(**args*, ***kwds*)

Dict subclass for counting hashable items. Sometimes called a bag or multiset. Elements are stored as dictionary keys and their counts are stored as dictionary values.

```
>>> c = Counter('abcdeabcdabcaba')  # count elements from a string
```

```
>>> c.most_common(3)                # three most common elements
[('a', 5), ('b', 4), ('c', 3)]
>>> sorted(c)                       # list all unique elements
['a', 'b', 'c', 'd', 'e']
>>> ''.join(sorted(c.elements()))   # list elements with repetitions
'aaaaabbbbcccdde'
```

```
>>> sum(c.values())                 # total of all counts
15
```

```
>>> c['a']                          # count of letter 'a'
5
>>> for elem in 'shazam':           # update counts from an iterable
...     c[elem] += 1                # by adding 1 to each element's count
>>> c['a']                          # now there are seven 'a'
7
>>> del c['b']                      # remove all 'b'
>>> c['b']                          # now there are zero 'b'
0
```

```
>>> d = Counter('simsalabim')       # make another counter
>>> c.update(d)                     # add in the second counter
>>> c['a']                          # now there are nine 'a'
9
```

```
>>> c.clear()                       # empty the counter
>>> c
Counter()
```

Note: If a count is set to zero or reduced to zero, it will remain in the counter until the entry is deleted or the counter is cleared:

```
>>> c = Counter('aaabbc')
>>> c['b'] -= 2                    # reduce the count of 'b' by two
>>> c.most_common()                # 'b' is still in, but its count is zero
[('a', 3), ('c', 1), ('b', 0)]
```

Note: Backported from Python 3.3. In Python 3.3 this is an alias.

class `xoutil.future.collections.`**`OrderedDict`**(**args*, ***kwds*)

Dictionary that remembers insertion order

Note: Backported from Python 3.3. In Python 3.3 this is an alias.

class `xoutil.future.collections.`**`OpenDictMixin`**

A mixin for mappings implementation that expose keys as attributes:

```
>>> from xoutil.objects import SafeDataItem as safe

>>> class MyOpenDict(OpenDictMixin, dict):
...     __slots__ = safe.slot(OpenDictMixin.__cache_name__, dict)

>>> d = MyOpenDict({'es': 'spanish'})
>>> d.es
'spanish'

>>> d['es'] = 'espanol'
>>> d.es
'espanol'
```

When setting or deleting an attribute, the attribute name is regarded as key in the mapping if neither of the following condition holds:

•The name is a *slot*.

•The object has a `__dict__` attribute and the name is key there.

This mixin defines the following features that can be redefined:

_key2identifier

Protected method, receive a key as argument and return a valid identifier that is used instead the key as an extended attribute.

__cache_name__

Inner field to store a cached mapping between actual keys and calculated attribute names. The field must be always implemented as a *SafeDataItem* descriptor and must be of type *dict*. There are two ways of implementing this:

•As a slot. The first time of this implementation is an example. Don't forget to pass the second parameter with the constructor *dict*.

•As a normal descriptor:

```
>>> from xoutil.objects import SafeDataItem as safe
>>> class MyOpenDict(OpenDictMixin, dict):
...     safe(OpenDictMixin.__cache_name__, dict)
```

Classes or Mixins that can be integrated with *dict* by inheritance must not have a *__slots__* definition. Because of that, this mixin must not declare any slot. If needed, it must be declared explicitly in customized classed like in the example in the first part of this documentation or in the definition of *opendict* class.

class `xoutil.future.collections.`**`OrderedSmartDict`**`(`**args*, ***kwds*`)`
A combination of the *OrderedDict* with the *SmartDictMixin*.

Warning: Initializing with kwargs does not ensure any initial ordering, since Python's keyword dict is not ordered. Use a list/tuple of pairs instead.

class `xoutil.future.collections.`**`SmartDictMixin`**
A mixin that extends the *update* method of dictionaries

Standard method allow only one positional argument, this allow several.

Note on using mixins in Python: method resolution order is calculated in the order of inheritance, if a mixin is defined to overwrite behavior already existent, use first that classes with it. See `SmartDict` below.

class `xoutil.future.collections.`**`StackedDict`**`(`**args*, ***kwargs*`)`
A multi-level mapping.

A level is entered by using the `push()` and is leaved by calling `pop()`.

The property `level` returns the actual number of levels.

When accessing keys they are searched from the latest level “upwards”, if such a key does not exists in any level a KeyError is raised.

Deleting a key only works in the *current level*; if it's not defined there a KeyError is raised. This means that you can't delete keys from the upper levels without `popping`.

Setting the value for key, sets it in the current level.

Changed in version 1.5.2: Based on the newly introduced `ChainMap`.

pop()

A deprecated alias for `pop_level()`.

Deprecated since version 1.7.0.

push(**args*, ***kwargs*)

A deprecated alias for `push_level()`.

Deprecated since version 1.7.0.

level

Return the current level number.

The first level is 0. Calling `push()` increases the current level (and returns it), while calling `pop()` decreases the current level (if possible).

peek()

Peeks the top level of the stack.

Returns a copy of the top-most level without any of the keys from lower levels.

Example:

```
>>> sdict = StackedDict(a=1, b=2)
>>> sdict.push(c=3)  # it returns the level...
1
>>> sdict.peek()
{'c': 3}
```

pop_level()

Pops the last pushed level and returns the whole level.

If there are no levels in the stacked dict, a TypeError is raised.

Returns A dict containing the poped level.

push_level(**args*, ***kwargs*)

Pushes a whole new level to the stacked dict.

Parameters

- **args** – Several mappings from which the new level will be initialled filled.
- **kwargs** – Values to fill the new level.

Returns The pushed `level` number.

class xoutil.future.collections.**ChainMap**(**maps*)

A ChainMap groups multiple dicts or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can accessed or updated using the maps attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A ChainMap incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.

All of the usual dictionary methods are supported. In addition, there is a maps attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps
A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(*m=None*)
Returns a new *ChainMap* containing a new map followed by all of the maps in the current instance. If `m` is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to `d.new_child()` is equivalent to: `ChainMap({}, *d.maps)`. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

Changed in version 1.5.5: The optional `m` parameter was added.

parents
Property returning a new ChainMap containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the nonlocal keyword used in nested scopes. A reference to `d.parents` is equivalent to: `ChainMap(*d.maps[1:])`.

Note: Backported from Python 3.4. In Python 3.4 this is an alias.

class `xoutil.future.collections.`**PascalSet**(**others*)
Collection of unique integer elements (implemented with intervals).

```
PascalSet(*others) -> new set object
```

New in version 1.7.1.

class `xoutil.future.collections.`**BitPascalSet**(**others*)
Collection of unique integer elements (implemented with bit-wise sets).

```
BitPascalSet(*others) -> new bit-set object
```

New in version 1.7.1.

`xoutil.future.datetime` - Basic date and time types

This module extends the standard library's `datetime`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

In Pytnon versions <= 3 date format fails for several dates, for example `date(1800, 1, 1).strftime("%Y")`. So, classes `date` and `datetime` are redefined if that case.

This problem could be solved by redefining the *strftime* function in the *time* module, because it is used for all *strftime* methods; but (WTF), Python double checks the year (in each method and then again in *time.strftime* function).

`xoutil.future.datetime.`**assure**(*obj*)
Make sure that a *date* or *datetime* instance is a safe version.

With safe it's meant that will use the adapted subclass on this module or the standard if these weren't generated.

Classes that could be assured are: *date*, *datetime*, *time* and *timedelta*.

We added the following features.

`xoutil.future.datetime.`**strfdelta**(*delta*)
Format a timedelta using a smart pretty algorithm.

Only two levels of values will be printed.

```
>>> def t(h, m):
...     return timedelta(hours=h, minutes=m)

>>> strfdelta(t(4, 56)) == '4h 56m'
True
```

`xoutil.future.datetime.`**`strftime`**`(`*dt*, *fmt*`)`

Used as *strftime* method of *date* and *datetime* redefined classes.

Also could be used with standard instances.

`xoutil.future.datetime.`**`get_month_first`**`(`*ref=None*`)`

Given a reference date, returns the first date of the same month. If *ref* is not given, then uses current date as the reference.

`xoutil.future.datetime.`**`get_month_last`**`(`*ref=None*`)`

Given a reference date, returns the last date of the same month. If *ref* is not given, then uses current date as the reference.

`xoutil.future.datetime.`**`get_next_month`**`(`*ref=None*, *lastday=False*`)`

Get the first or last day of the *next month*.

If *lastday* is False return the first date of the *next month*. Otherwise, return the last date.

The *next month* is computed with regards to a reference date. If *ref* is None, take the current date as the reference.

Examples:

```
>>> get_next_month(date(2017, 1, 23))
date(2017, 2, 1)
```

```
>>> get_next_month(date(2017, 1, 23), lastday=True)
date(2017, 2, 28)
```

New in version 1.7.3.

`xoutil.future.datetime.`**`is_full_month`**`(`*start*, *end*`)`

Returns true if the arguments comprises a whole month.

class `xoutil.future.datetime.`**`flextime`**

`xoutil.future.datetime.`**`daterange`**`(`[*start*], *stop*[, *step*]`)`

Similar to standard 'range' function, but for date objets.

Returns an iterator that yields each date in the range of `[start, stop)`, not including the stop.

If *start* is given, it must be a date (or *datetime*) value; and in this case only *stop* may be an integer meaning the numbers of days to look ahead (or back if *stop* is negative).

If only *stop* is given, *start* will be the first day of stop's month.

step, if given, should be a non-zero integer meaning the numbers of days to jump from one date to the next. It defaults to `1`. If it's positive then *stop* should happen after *start*, otherwise no dates will be yielded. If it's negative *stop* should be before *start*.

As with *range*, *stop* is never included in the yielded dates.

class `xoutil.future.datetime.`**`DateField`**`(`*name*, *nullable=False*`)`

A simple descriptor for dates.

Ensures that assigned values must be parseable dates and parses them.

class `xoutil.future.datetime.`**`TimeSpan`**(*start_date=None*, *end_date=None*)

A *continuous* span of time.

Time spans objects are iterable. They yield exactly two times: first the start date, and then the end date:

```
>>> ts = TimeSpan('2017-08-01', '2017-09-01')
>>> tuple(ts)
(date(2017, 8, 1), date(2017, 9, 1))
```

Time spans objects have two items:

```
>>> ts[0]
date(2017, 8, 1)

>>> ts[1]
date(2017, 9, 1)

>>> ts[:]
(date(2017, 8, 1), date(2017, 9, 1))
```

Two time spans are equal if their start_date and end_date are equal. When comparing a time span with a date, the date is coerced to a time span (`from_date()`).

A time span with its *start* set to None is unbound to the past. A time span with its *end* set to None is unbound to the future. A time span that is both unbound to the past and the future contains all possible dates. A time span that is not unbound in any direction is `bound` .

A bound time span is `valid` if its start date comes before its end date.

Time spans can `intersect`, compared for containment of dates and by the subset/superset order operations (<=, >=). In this regard, they represent the *set* of dates between *start* and *end*, inclusively.

Warning: Time spans don't implement the union or difference operations expected in sets because the difference/union of two span is not necessarily *continuous*.

classmethod **`from_date`**(*date*)

Return a new time span that covers a single *date*.

`past_unbound`

True if the time span is not bound into the past.

`future_unbound`

True if the time span is not bound into the future.

`unbound`

True if the time span is `unbound into the past` or `unbount into the future` or both.

`bound`

True if the time span is not `unbound`.

`valid`

A bound time span is valid if it starts before it ends.

Unbound time spans are always valid.

`__le__`(*other*)

True if *other* is a superset.

`issubset`()

An alias for `__le__()`.

__ge__(*other*)
True if *other* is a subset.

issuperset()
An alias for `__ge__()`.

covers()
An alias for `__ge__()`.

isdisjoint(*other*)

overlaps(*other*)
Test if the time spans overlaps.

__and__(*other*)
Get the time span that is the intersection with another time span.

If two time spans don't overlap, return the `empty time span`.

If *other* is not a TimeSpan we try to create one. If *other* is a date, we create the TimeSpan that starts and end that very day. Other types are passed unchanged to the constructor.

__mul__()
An alias for `__and__()`.

intersection(**others*)
Return `self [& other1 & ...]`.

EmptyTimeSpan
The empty time span. It's not an instance of `TimeSpan` but engage set-like operations: union, intersection, etc.

No date is a member of the empty time span. The empty time span is a proper subset of any time span. It's only a superset of itself. It's not a proper superset of any other time span nor itself.

This instance is a singleton. However, if you pickle it with protocol 1 and unpickle it, you'll lose that property. It's best to test with the equality operator ==.

`xoutil.future.functools` - Higher-order functions and callable objects

This module extends the standard library's `functools`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

class `xoutil.future.functools.`**ctuple**
Simple tuple marker for `compose()`.

Since is a callable you may use it directly in `compose` instead of changing your functions to returns ctuples instead of tuples:

```
>>> def compat_print(*args):
...     for arg in args:
...         print(arg)

>>> compose(compat_print, ctuple, list, range, math=False)(3)
0
1
2

# Without ctuple prints the list
```

```
>>> compose(compat_print, list, range, math=False)(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`xoutil.future.functools.`**`compose`**`(`**funcs*, *math=True*`)`

Returns a function that is the composition of several *callables*.

By default *compose* behaves like mathematical function composition: this is to say that `compose(f1, ... fn)` is equivalent to `lambda _x: fn(...(f1(_x))...)`.

If any "intermediate" function returns a *`ctuple`* it is expanded as several positional arguments to the next function.

Changed in version 1.5.5: At least a callable must be passed, otherwise a TypeError is raised. If a single callable is passed it is returned without change.

Parameters `math` – Indicates if *compose* should behave like mathematical function composition: last function in *funcs* is applied last. If False, then the last function in *func* is applied first.

`xoutil.future.functools.`**`power`**`(`**funcs*, *times*`)`

Returns the "power" composition of several functions.

Examples:

```
>>> import operator
>>> f = power(partial(operator.mul, 3), 3)
>>> f(23) == 3*(3*(3*23))
True

>>> power(operator.neg)
Traceback (most recent call last):
...
TypeError: power() takes at least 2 arguments (1 given)
```

class `xoutil.future.functools.`**`lwraps`**`(`*f*, *n*, *, *name=None*, *doc=None*, *wrapped=None*`)`

Lambda wrapper.

Useful for decorate lambda functions with name and documentation.

As positional arguments could be passed the function to be decorated and the name in any order. So the next two `identity` definitions are equivalents:

```
>>> from xoutil.future.functools import lwraps as lw

>>> identity = lw('identity', lambda arg: arg)

>>> identity = lw(lambda arg: arg, 'identity')
```

As keyword arguments could be passed some special values, and any number of literal values to be assigned:

- **name**: The name of the function (`__name__`); only valid if not given as positional argument.
- **doc**: The documentation (`__doc__` field).
- **wrapped**: An object to extract all values not yet assigned. These values are ('__module__', '__name__' and '__doc__') to be assigned, and '__dict__' to be updated.

If the function to decorate is present in the positional arguments, this same argument function is directly returned after decorated; if not a decorator is returned similar to standard `wraps()`.

For example:

```
>>> from xoutil.future.functools import lwraps as lw

>>> is_valid_age = lw('is-valid-human-age', lambda age: 0 < age <= 120,
...                   doc=('A predicate to evaluate if an age is '
...                        'valid for a human being.')

>>> @lw(wrapped=is_valid_age)
... def is_valid_working_age(age):
...     return 18 < age <= 70

>>> is_valid_age(16)
True

>>> is_valid_age(200)
False

>>> is_valid_working_age(16)
False
```

New in version 1.7.0.

`xoutil.future.functools.`**`curry`**(*f*)

Return a function that automatically 'curries' is positional arguments.

Example:

```
>>> add = curry(lambda x, y: x + y)
>>> add(1)(2)
3

>>> add(1, 2)
3

>>> add()()()(1, 2)
3
```

We have backported several Python 3.3 features but maybe not all.

`xoutil.future.functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a wrapper function to look like the wrapped function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants *WRAPPER_ASSIGNMENTS* (which assigns to the wrapper function's *__name__*, *__module__*, *__annotations__* and *__doc__*, the documentation string) and *WRAPPER_UPDATES* (which updates the wrapper function's *__dict__*, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as `lru_cache()`), this function automatically adds a *__wrapped__* attribute to the wrapper that refers to the original function.

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`update_wrapper()` may be used with callables other than functions. Any attributes named in assigned or updated that are missing from the object being wrapped are ignored (i.e. this function will not attempt to

set them on the wrapper function). AttributeError is still raised if the wrapper function itself is missing any attributes named in updated.

`xoutil.future.inspect` - Inspect live objects

This module extends the standard library's `functools`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

`xoutil.future.inspect.`**`get_attr_value`**`(`*obj*, *name*, **default*`)`

Get a named attribute from an object in a safe way.

Similar to *getattr* but without triggering dynamic look-up via the descriptor protocol, *__getattr__* or *__getattribute__* by using `getattr_static()`.

`xoutil.future.inspect.`**`type_name`**`(`**args*, ***kw*`)`

Return the internal name for a type or a callable.

This function is safe. If :param obj: is not an instance of a proper type then returns the following depending on :param affirm:

- If `False` returns None.
- If `True` convert a single object to its type before returns the name, but if is a tuple, list or set; returns a string with a representation of contained types.

Examples:

```
>>> safe_name(int)
'int'

>>> safe_name(0) is None
True

>>> safe_name(0, affirm=True)
'int'

>>> safe_name((0, 1.1)) is None
True

>>> safe_name((0, 1.1), affirm=True)
'(int, float)'
```

We have backported several Python 3.3 features but maybe not all (some protected structures are not presented in this documentation).

`xoutil.future.inspect.`**`getfullargspec`**`(`*func*`)`

`xoutil.future.inspect.`**`getattr_static`**`(`*obj*, *attr*, *default=<object object>*`)`

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptor objects instead of instance members in some cases. See the documentation for details.

`xoutil.future.json` - Encode and decode the JSON format

This module extends the standard library's `json`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

class `xoutil.future.json.`**`JSONEncoder`**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Extensible JSON <http://json.org> encoder for Python data structures.

Supports the following objects and types by default:

Python	JSON
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
None	null

To extend this to recognize other objects, subclass and implement a `.default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

Xoutil extends this class by supporting the following data-types:

- *datetime*, *date* and *time* values, which are translated to strings using ISO format.
- *Decimal* values, which are represented as a string representation.
- Iterables, which are represented as lists.

`xoutil.future.json.`**`encode_string`**(*string*, *ensure_ascii=True*)

Return a JSON representation of a Python string.

Parameters **`ensure_ascii`** – If True, the output is guaranteed to be of type *str* with all incoming non-ASCII characters escaped. If False, the output can contain non-ASCII characters.

`xoutil.future.pprint` - Extension to the data pretty printer

This modules includes all the Python's standard library features in module `pprint` and adds the function *`ppformat()`*, which just returns a string of the pretty-formatted object.

New in version 1.4.1.

`xoutil.future.pprint.`**`ppformat`**(*obj*)

Just like `pprint()` but always returning a result.

Returns The pretty formated text.

Return type *unicode* in Python 2, *str* in Python 3.

`xoutil.future.subprocess` - Extensions to *subprocess* stardard module

New in version 1.2.1.

This module contains extensions to the `subprocess` standard library module. It may be used as a replacement of the standard.

`xoutil.future.subprocess.`**`call_and_check_output`**(*args*, *, *stdin=None*, *shell=False*)

This function combines the result of both *call* and *check_output* (from the standard library module).

Returns a tuple `(retcode, output, err_output)`.

`xoutil.future.textwrap` - Text wrapping and filling

This module extends the standard library's `textwrap`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

`xoutil.future.textwrap.`**`dedent`**(*text*, *skip_firstline=False*)

Remove any common leading whitespace from every line in text.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `"    hello"` and `"\thello"` are considered to have no common leading whitespace.

If *skip_firstline* is True, the first line is separated from the rest of the body. This helps with docstrings that follow **PEP 257**.

Warning: The *skip_firstline* argument is missing in standard library.

We have backported several Python 3.3 features but maybe not all.

`xoutil.future.textwrap.`**`indent`**(*text*, *prefix*, *predicate=None*)

Adds 'prefix' to the beginning of selected lines in 'text'.

If 'predicate' is provided, 'prefix' will only be added to the lines where 'predicate(line)' is True. If 'predicate' is not provided, it will default to adding 'prefix' to all non-empty lines that do not consist solely of whitespace characters.

Note: Backported from Python 3.3. In Python 3.3 this is an alias.

`xoutil.future.threading` - Higher-level threading interface

This module extends the standard library's `threading`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added the following features.

`xoutil.future.threading.`**`async_call`**(*func*, *args=None*, *kwargs=None*, *callback=None*, *onerror=None*)

Executes a function asynchronously.

The function receives the given positional and keyword arguments

If *callback* is provided, it is called with a single positional argument: the result of calling *func(*args, **kwargs)*.

If the called function ends with an exception and *onerror* is provided, it is called with the exception object.

Returns An event object that gets signalled when the function ends its execution whether normally or with an error.

Return type `Event`

`xoutil.future.threading.`**`sync_call`**(*funcs*, *callback*, *timeout=None*)

Calls several functions, each one in it's own thread.

Waits for all to end.

Each time a function ends the *callback* is called (wrapped in a lock to avoid race conditions) with the result of the as a single positional argument.

If *timeout* is not None it sould be a float number indicading the seconds to wait before aborting. Functions that terminated before the timeout will have called *callback*, but those that are still working will be ignored.

Todo

Abort the execution of a thread.

Parameters **`funcs`** – A sequences of callables that receive no arguments.

`xoutil.future.types` - Names for built-in types and extensions

This module extends the standard library's `functools`. You may use it as a drop-in replacement in many cases.

Avoid importing `*` from this module since could be different in Python 2.7 and Python 3.3.

We added mainly compatibility type definitions, those that each one could be in one version and not in other.

`xoutil.future.types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Create a class object dynamically using the appropriate metaclass.

New in version 1.5.5.

`xoutil.future.types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Call the __prepare__ method of the appropriate metaclass.

Returns (metaclass, namespace, kwds) as a 3-tuple

metaclass is the appropriate metaclass *namespace* is the prepared class namespace *kwds* is an updated copy of the passed in kwds argument with any 'metaclass' entry removed. If no kwds argument is passed in, this will be an empty dict.

New in version 1.5.5.

`xoutil.future.types.`**`DictProxyType`**

alias of `dictproxy`

class `xoutil.future.types.`**`MappingProxyType`**

New in version 1.5.5.

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

Note: In Python 3.3+ this is an alias for `types.MappingProxyType` in the standard library.

---

class `xoutil.future.types.`**`SimpleNamespace`**

New in version 1.5.5.

A simple `object` subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike `object`, with `SimpleNamespace` you can add and remove attributes. If a `SimpleNamespace` object is initialized with keyword arguments, those are directly added to the underlying namespace.

The type is roughly equivalent to the following code:

```
class SimpleNamespace(object):
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))
    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use `namedtuple()` instead.

Note: In Python 3.4+ this is an alias to `types.SimpleNamespace`.

class `xoutil.future.types.`**`DynamicClassAttribute`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to `__getattr__()`.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class's `__getattr__()` method; this is done by raising `AttributeError`.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see `Enum` for an example).

New in version 1.5.5.

Changed in version 1.8.0: Inherits from *property*

Note: The class *Enum* mentioned has not yet been back-ported.

Note: In Python version>=3.4 this is an alias to `types.DynamicClassAttribute`.

`xoutil.html` – Helpers for manipulating HTML

Deprecated since version 1.8.0. This module defines utilities to manipulate HTML.

This module backports several utilities from Python 3.2.

Because now we deprecated it, we moved here documentation to remove it in one shot.

xoutil.html.entities – Definitions of HTML general entities

This module defines tree dictionaries, `name2codepoint`, `codepoint2name`, and `entitydefs`.

`entitydefs` is used to provide the *entitydefs* attribute of the *`xoutil.html.parser.HTMLParser`* class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

`xoutil.html.`**`entitydefs`**
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

`xoutil.html.`**`name2codepoint`**
A dictionary that maps HTML entity names to the Unicode codepoints.

`xoutil.html.`**`codepoint2name`**
A dictionary that maps Unicode codepoints to HTML entity names

xoutil.html.parser – A simple parser that can handle HTML and XHTML

This module defines a class HTMLParser which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

Warning: This module has not being made Python 2.7 and 3.2 compatible.

class `xoutil.html.`**`HTMLParser`**(*strict=True*)
Create a parser instance. If strict is True (the default), invalid HTML results in *`HTMLParseError`* exceptions [1]. If strict is False, the parser uses heuristics to make a best guess at the intention of any invalid HTML it encounters, similar to the way most browsers do. Using strict=False is advised.

An :class'HTMLParser' instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass HTMLParser and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Changed in version 3.2: strict keyword added

class `xoutil.html.`**`HTMLParseError`**
Exception raised by the *`HTMLParser`* class when it encounters an error while parsing and strict is True. This exception provides three attributes: msg is a brief message explaining the error, lineno is the number of the line on which the broken construct was detected, and offset is the number of characters into the line at which the construct starts.

`xoutil.html.`**`escape`**(*s*, *quote=True*)
Replace special characters "&", "<" and ">" to HTML-safe sequences

If the optional flag quote is true (the default), the quotation mark characters, both double quote (") and single quote (') characters are also translated.

Sub-modules on this package

`xoutil.html.entities` – Definitions of HTML general entities

This module defines tree dictionaries, `name2codepoint`, `codepoint2name`, and `entitydefs`.

`entitydefs` is used to provide the *entitydefs* attribute of the `xoutil.html.parser.HTMLParser` class. The definition provided here contains all the entities defined by XHTML 1.0 that can be handled using simple textual substitution in the Latin-1 character set (ISO-8859-1).

`xoutil.html.entities.`**`entitydefs`**
A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

`xoutil.html.entities.`**`name2codepoint`**
A dictionary that maps HTML entity names to the Unicode codepoints.

`xoutil.html.entities.`**`codepoint2name`**
A dictionary that maps Unicode codepoints to HTML entity names

`xoutil.html.parser` – A simple parser that can handle HTML and XHTML

This module defines a class HTMLParser which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

Warning: This module has not being made Python 2.7 and 3.2 compatible.

class `xoutil.html.parser.`**`HTMLParser`**`(`*strict=True*`)`
Create a parser instance. If strict is True (the default), invalid HTML results in `HTMLParseError` exceptions [1]. If strict is False, the parser uses heuristics to make a best guess at the intention of any invalid HTML it encounters, similar to the way most browsers do. Using strict=False is advised.

An :class‘HTMLParser‘ instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass HTMLParser and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Changed in version 3.2: strict keyword added

class `xoutil.html.parser.`**`HTMLParseError`**
Exception raised by the `HTMLParser` class when it encounters an error while parsing and strict is True. This exception provides three attributes: msg is a brief message explaining the error, lineno is the number of the line on which the broken construct was detected, and offset is the number of characters into the line at which the construct starts.

`xoutil.infinity` - An infinite value

`xoutil.infinity.Infinity`
The positive infinite value. The negative infinite value is `-Infinity`.

These values are only sensible for comparison. Arithmetic is not supported.

The type of values that is comparable with `Infinity` is controlled by the ABC `InfinityComparable`.

class `xoutil.infinity.`**`InfinityComparable`**
Any type that can be sensibly compared to infinity.

All types in the `number` tower are *always* comparable.

Classes `datetime.date`, `datetime.datetime`, and `datetime.timedelta` are automatically registered.

`xoutil.iterators` - Functions creating iterators for efficient looping

Several util functions for iterators

`xoutil.iterators.`**`dict_update_new`**`(`*target*, *source*, *fail=False*`)`

Update values in *source* that are new (not present) in *target*.

If *fail* is True and a value is already set, an error is raised.

`xoutil.iterators.`**`first_n`**`(`*iterable*, *n=1*, *fill=Unset*`)`

Takes the first *n* items from iterable.

If there are less than *n* items in the iterable and *fill* is `Unset`, a StopIteration exception is raised; otherwise it's used as a filling pattern as explained below.

Parameters

- **`iterable`** – An iterable from which the first *n* items should be collected.
- **`n`** (*`int`*) – The number of items to collect
- **`fill`** – The filling pattern to use. It may be:
  - a collection, in which case *first_n* fills the last items by cycling over *fill*.
  - anything else is used as the filling pattern by repeating.

Returns The first *n* items from *iterable*, probably with a filling pattern at the end.

Return type generator object

New in version 1.2.0.

Changed in version 1.4.0: The notion of collection for the *fill* argument uses `xoutil.types.is_collection()` instead of probing for the `__iter__` method.

Changed in version 1.7.2: The notion of collection for the *fill* argument uses `isinstance(fill, Iterable)` replacing `xoutil.types.is_collection()`. We must be consistent with *iterable* argument that allow an string as a valid iterable and *is_collection* not.

`xoutil.iterators.`**`first_non_null`**`(`*iterable*, *default=None*`)`

Returns the first value from iterable which is non-null.

This is roughly the same as:

```
next((x for x in iter(iterable) if x), default)
```

New in version 1.4.0.

`xoutil.iterators.`**`slides`**`(`*iterable*, *width=2*, *fill=None*`)`

Creates a sliding window of a given *width* over an iterable:

```
>>> list(slides(range(1, 11)))
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

If the iterator does not yield a width-aligned number of items, the last slice returned is filled with *fill* (by default None):

```
>>> list(slides(range(1, 11), width=3))
[(1, 2, 3), (4, 5, 6), (7, 8, 9), (10, None, None)]
```

Changed in version 1.4.0: If the *fill* argument is a collection is cycled over to get the filling, just like in `first_n()`.

Changed in version 1.4.2: The *fill* argument now defaults to None, instead of Unset.

`xoutil.iterators.`**`continuously_slides`**`(`*iterable*, *width=2*, *fill=None*`)`

Similar to `slides()` but moves one item at the time (i.e continuously).

fill is only used to fill the fist chunk if the *iterable* has less items than the *width* of the window.

Example (generate a texts tri-grams):

```
>>> slider = continuously_slides(str('maupassant'), 3)
>>> list(str('').join(chunk) for chunk in slider)
['mau', 'aup', 'upa', 'pas', 'ass', 'ssa', 'san', 'ant']
```

`xoutil.iterators.`**`ungroup`**`(`*iterator*`)`

Reverses the operation of `itertools.groupby()` (or similar).

The *iterator* should produce pairs of `(_, xs)`; where `xs` is another iterator (or iterable).

It's guaranteed that the *iterator* will be consumed at the *boundaries* of each pair, i.e. before taking another pair `(_, ys)` from *iterator* the first `xs` will be fully yielded.

Demonstration:

```
>>> def groups():
...     def chunk(s):
...         for x in range(s, s+3):
...             print('Yielding x:', x)
...             yield x
...
...     for g in range(2):
...         print('Yielding group', g)
...         yield g, chunk(g)
```

```
>>> list(ungroup(groups()))
Yielding group 0
Yielding x: 0
Yielding x: 1
Yielding x: 2
Yielding group 1
Yielding x: 1
Yielding x: 2
Yielding x: 3
[0, 1, 2, 1, 2, 3]
```

This is not the same as:

```
>>> import itertools
>>> xs = itertools.chain(*(xs for _, xs in groups()))
Yielding group 0
Yielding group 1
```

Notice that the iterator was fully consumed just to create the arguments to `chain()`.

New in version 1.7.3.

`xoutil.iterators.`**`delete_duplicates`**`(`*seq*[, *key=lambda x: x*]`)`

Remove all duplicate elements from *seq*.

Two items x and y are considered equal (duplicates) if `key(x) == key(y)`. By default *key* is the identity function.

Works with any sequence that supports `len()`, `__getitem__()`, and `addition`.

Note: `seq.__getitem__` should work properly with slices.

The return type will be the same as that of the original sequence.

New in version 1.5.5.

Changed in version 1.7.4: Added the *key* argument. Clarified the documentation: *seq* should also implement the `__add__` method and that its `__getitem__` method should deal with slices.

`xoutil.iterators.`**`iter_delete_duplicates`**(*iter*[, *key=lambda x: x*])

Yields non-repeating items from *iter*.

key has the same meaning as in `delete_duplicates()`.

Examples:

```
>>> list(iter_delete_duplicates('AAAaBBBA'))
['A', 'a', 'B', 'A']
```

```
>>> list(iter_delete_duplicates('AAAaBBBA', key=lambda x: x.lower()))
['A', 'B', 'A']
```

New in version 1.7.4.

`xoutil.iterators.`**`fake_dict_iteritems`**(*source*)

Iterate (key, value) in a source fake mapping.

A fake mapping must define at least methods *keys* and `__getitem__()`.

Warning: Deprecated since 1.7.0. This was actually in risk since 1.4.0.

`xoutil.iterators.`**`flatten`**(*sequence*, *is_scalar=xoutil.types.is_scalar*, *depth=None*)

Flatten-out a sequence.

It takes care of everything deemed a collection (i.e, not a scalar according to the callable passed in *is_scalar* argument; if None, `xoutil.types.is_scalar()` is assumed):

```
>>> from xoutil.eight import range
>>> range_ = lambda *a: list(range(*a))
>>> tuple(flatten((1, range_(2, 5), range(5, 10))))
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

If *depth* is None the collection is flattened recursiverly until the “bottom” is reached. If *depth* is an integer then the collection is flattened up to that level. *depth=0* means not to flatten. Nested iterators are not “exploded” if under the stated *depth*:

```
# In the following doctest we use ``...range(...X)`` because the
# string repr of range differs in Py2 and Py3k.

>>> tuple(flatten((range_(2), range(2, 4)), depth=0))  # doctest: +ELLIPSIS
([0, 1], ...range(2, 4))
```

```
>>> tuple(flatten((range(2), range_(2, 4)), depth=0))  # doctest: +ELLIPSIS
(...range(...2), [2, 3])
```

`xoutil.iterators.`**`zip`**([*iter1*[, *iter2*[, ...]]])
: Return a zip-like object whose *next()* method returns a tuple where the i-th element comes from the i-th iterable argument. The *next()* method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration.

  This method is actually the standard `itertools.izip()` when in Python 2.7, and the builtin `zip` when in Python 3.

`xoutil.iterators.`**`map`**(*func*, **iterables*)
: Make an iterator that computes the function using arguments from each of the iterables. It stops when the shortest iterable is exhausted instead of filling in None for shorter iterables.

  This method is actually the stardard `itertools.imap` when in Python 2.7, and the builtin `map` when in Python 3.

`xoutil.iterators.`**`zip_longest`**(**iterables*, *fillvalue=None*)
: Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

  If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`). If not specified, *fillvalue* defaults to None.

  This function is actually an alias to `itertools.izip_longest()` in Python 2.7, and an alias to `itertools.zip_longest()` in Python 3.3.

`xoutil.keywords` – Tools for manage Python keywords as names

Tools for manage Python keywords as names.

Reserved Python keywords can't be used as attribute names, so this module functions use the convention of rename the name using an underscore as suffix when a reserved keyword is used as name.

`xoutil.keywords.`**`delkwd`**(*obj*, *name*)
: Like *delattr* but taking into account Python keywords.

`xoutil.keywords.`**`getkwd`**(*obj*, *name*, *default=None*)
: Like *getattr* but taking into account Python keywords.

`xoutil.keywords.`**`kwd_deleter`**(*obj*)
: partial(delkwd, obj)

`xoutil.keywords.`**`kwd_getter`**(*obj*)
: partial(getkwd, obj)

`xoutil.keywords.`**`kwd_setter`**(*obj*)
: partial(setkwd, obj)

`xoutil.keywords.`**`org_kwd`**(*name*)
: Remove the underscore suffix if name starts with a Python keyword.

`xoutil.keywords.`**`setkwd`**(*obj*, *name*, *value*)
: Like *setattr* but taking into account Python keywords.

`xoutil.keywords.`**`suffix_kwd`**(*name*)
: Add an underscore suffix if name if a Python keyword.

`xoutil.logger` - Standard logger helpers

Usage:

```
logger.debug('Some debug message')
```

Basically you may request any of the loggers attribute/method and this module will return the logger's attribute corresponding to the loggers of the calling module. This avoids the boilerplate seen in most codes:

```
logger = logging.getLogger(__name__)
```

You may simply do:

```
from xoutil.logger import debug
debug('Some debug message')
```

The proper logger will be selected by this module.

Note: Notice this won't configure any handler for you. Only the calling pattern is affected. You must configure your loggers as usual.

`xoutil.modules` – Utilities for working with modules

Modules utilities.

`xoutil.modules.`**`copy_members`**`(`*source=None*, *target=None*`)`

Copy module members from *source* to *target*.

It's common in *xoutil* package to extend Python modules with the same name, for example *xoutil.datetime* has all public members of Python's *datetime*. `copy_members()` can be used to copy all members from the original module to the extended one.

Parameters

- **`source`** – string with source module name or module itself.

  If not given, is assumed as the last module part name of *target*.

- **`target`** – string with target module name or module itself.

  If not given, target name is looked in the stack of caller module.

Returns Source module.

Return type *ModuleType*

Warning: Implementation detail

Function used to inspect the stack is not guaranteed to exist in all implementations of Python.

`xoutil.modules.`**`customize`**`(`*module*, *custom_attrs=None*, *meta=None*`)`

Replaces a *module* by a custom one.

Injects all kwargs into the newly created module's class. This allows to have module into which we may have properties or other type of descriptors.

Parameters

- **`module`** – The module object to customize.
- **`custom_attrs`** – A dictionary of custom attributes that should be injected in the customized module.

  New in version 1.4.2: Changes the API, no longer uses the `**kwargs` idiom for custom attributes.
- **`meta`** – The metaclass of the module type. This should be a subclass of *type*. We will actually subclass this metaclass to properly inject *custom_attrs* in our own internal metaclass.

Returns A tuple of `(module, customized, class)` with the module in the first place, *customized* will be True only if the module was created (i.e `customize()` is idempotent), and the third item will be the class of the module (the first item).

`xoutil.modules.`**`force_module`**`(`*ref=None*`)`

Load a module from a string or return module if already created.

If *ref* is not specified (or integer) calling module is assumed looking in the stack.

Note: Implementation detail

Function used to inspect the stack is not guaranteed to exist in all implementations of Python.

`xoutil.modules.`**`get_module_path`**`(`*module*`)`

Gets the absolute path of a *module*.

Parameters **`module`** – Either module object or a (dotted) string for the module.

Returns The path of the module.

If the module is a package, returns the directory path (not the path to the `__init__`).

If *module* is a string and it's not absolute, raises a TypeError.

`xoutil.modules.`**`modulemethod`**`(`*func*`)`

Decorator that defines a module-level method.

Simply a module-level method, will always receive a first argument *self* with the module object.

`xoutil.modules.`**`moduleproperty`**`(`*getter, setter=None, deleter=None, doc=None, base=<type 'property'>*`)`

Decorator that creates a module-level property.

The module of the *getter* is replaced by a custom implementation of the module, and the property is injected to the custom module's class.

The parameter *base* serves the purpose of changing the base for the property. For instance, this allows you to have `memoized_properties` at the module-level:

```
def memoized(self):
    return self
memoized = moduleproperty(memoized, base=memoized_property)
```

`xoutil.names` – Utilities for handling objects names

A protocol to obtain or manage object names.

`xoutil.names.`**`nameof`**`(`**objects*, *depth=1*, *inner=False*, *typed=False*, *full=False*, *safe=False*`)`

Obtain the name of each one of a set of objects.

New in version 1.4.0.

> Changed in version 1.6.0: Keyword arguments are now keyword-only arguments. Support for several objects Improved the semantics of parameter *full*. Added the *safe* keyword argument.

If no object is given, None is returned; if only one object is given, a single string is returned; otherwise a list of strings is returned.

The name of an object is normally the variable name in the calling stack.

If the object is not present calling frame, up to five frame levels are searched. Use the *depth* keyword argument to specify a different starting point and the search will proceed five levels from this frame up.

If the same object has several good names a single one is arbitrarily chosen.

Good names candidates are retrieved based on the keywords arguments *full*, *inner*, *safe* and *typed*.

If *typed* is True and the object is not a type name or a callable (see `xoutil.future.inspect.safe_name()`), then the *type* of the object is used instead.

If *inner* is True we try to extract the name by introspection instead of looking for the object in the frame stack.

If *full* is True the full identifier of the object is preferred. In this case if *inner* is False the local-name for the object is found. If *inner* is True, find the import-name.

If *safe* is True, returned value is converted -if it is not- into a valid Python identifier, though you should not trust this identifier resolves to the value.

See *the examples in the documentation*.

`xoutil.names.`**`identifier_from`**`(`*obj*`)`

Build an valid identifier from the name extracted from an object.

New in version 1.5.6.

First, check if argument is a type and then returns the name of the type prefixed with _ if valid; otherwise calls *nameof* function repeatedly until a valid identifier is found using the following order logic: `inner=True`, without arguments looking-up a variable in the calling stack, and `typed=True`. Returns None if no valid value is found.

Examples:

```
>>> identifier_from({})
'dict'
```

Use cases for getting the name of an object

The function `nameof()` is useful for cases when you get a value and you need a name. This is a common need when doing framework-level code that tries to avoid repetition of concepts.

Solutions with `nameof()`

Properly calculate the tasks' name in Celery applications

Celery warns about how to import the tasks. If in a module you import your task using an absolute import, and in another module you import it using a relative import, Celery regards them as different tasks. You must either use a consistent import style, or give a name for the task. Using *nameof* you can easily fix this problem.

Assume you create a `celapp.tasks.basic` module with this code:

```
>>> def celery_task(celeryapp, *args, **kwargs):
...    def decorator(func):
...        from xoutil.names import nameof
...        taskname = nameof(func, full=True, inner=True)
...        return celeryapp.task(name=taskname, *args, **kwargs)(func)
...    return decorator

>>> from celery import Celery
>>> app = Celery()
>>> @celery_task(app)
... def add(x, y):
...     return x + y
```

Then importing the task directly in a shell will have the correct name:

```
>>> from celapp.tasks.basic import add
>>> add.name
'celapp.tasks.basic.add'
```

Another module that imports the task will also see the proper name. Say you have the module `celapp.consumer`:

```
>>> from .tasks import basic

>>> def get_name(taskname):
...     task = getattr(basic, taskname)
...     return task.name
```

Then:

```
>>> from celapp.consumer import get_name
>>> get_name('add')
'celapp.tasks.basic.add'
```

Despite that you imported the `basic` module with a relative import the name is fully calculated.

`xoutil.objects` - Functions for dealing with objects

Several utilities for objects in general.

`xoutil.objects.`**`validate_attrs`**(*source*, *target*, *force_equals=()*, *force_differents=()*)

Makes a ‘comparison’ of *source* and *target* by its attributes (or keys).

This function returns True if and only if both of these tests pass:

- All attributes in *force_equals* are equal in *source* and *target*
- All attributes in *force_differents* are different in *source* and *target*

For instance:

```
>>> class Person(object):
...     def __init__(self, **kwargs):
...         for which in kwargs:
...             setattr(self, which, kwargs[which])

>>> source = Person(name='Manuel', age=33, sex='male')
>>> target = {'name': 'Manuel', 'age': 4, 'sex': 'male'}

>>> validate_attrs(source, target, force_equals=('sex',),
...                force_differents=('age',))
True

>>> validate_attrs(source, target, force_equals=('age',))
False
```

If both *force_equals* and *force_differents* are empty it will return True:

```
>>> validate_attrs(source, target)
True
```

`xoutil.objects.`**`iterate_over`**(*source*, **keys*)
Yields pairs of (key, value) for of all *keys* in *source*.

If any *key* is missing from *source* is ignored (not yielded).

If *source* is a `collection`, iterate over each of the items searching for any of keys. This is not recursive.

If no *keys* are provided, return an "empty" iterator – i.e will raise StopIteration upon calling *next*.

New in version 1.5.2.

`xoutil.objects.`**`smart_getter`**(*obj*, *strict=False*)
Returns a smart getter for *obj*.

If *obj* is a mapping, it returns the `.get()` method bound to the object *obj*, otherwise it returns a partial of `getattr` on *obj*.

Parameters **`strict`** – Set this to True so that the returned getter checks that keys/attrs exists. If *strict* is True the getter may raise a KeyError or an AttributeError.

Changed in version 1.5.3: Added the parameter *strict*.

`xoutil.objects.`**`smart_getter_and_deleter`**(*obj*)
Returns a function that get and deletes either a key or an attribute of obj depending on the type of *obj*.

If *obj* is a *collections.Mapping* it must be a *collections.MutableMapping*.

`xoutil.objects.`**`popattr`**(*obj*, *name*, *default=None*)
Looks for an attribute in the *obj* and returns its value and removes the attribute. If the attribute is not found, *default* is returned instead.

Examples:

```
>>> class Foo(object):
...   a = 1
>>> foo = Foo()
>>> foo.a = 2
>>> popattr(foo, 'a')
2
>>> popattr(foo, 'a')
1
>>> popattr(foo, 'a') is None
True
```

`xoutil.objects.`**`setdefaultattr`**(*obj*, *name*, *value*)
Sets the attribute name to value if it is not set:

```
>>> class Someclass(object): pass
>>> inst = Someclass()
>>> setdefaultattr(inst, 'foo', 'bar')
'bar'

>>> inst.foo
'bar'

>>> inst.spam = 'egg'
>>> setdefaultattr(inst, 'spam', 'with ham')
'egg'
```

(*New in version 1.2.1*). If you want the value to be lazily evaluated you may provide a lazy-lambda:

```
>>> inst = Someclass()
>>> inst.a = 1
>>> def setting_a():
...     print('Evaluating!')
...     return 'a'

>>> setdefaultattr(inst, 'a', lazy(setting_a))
1

>>> setdefaultattr(inst, 'ab', lazy(setting_a))
Evaluating!
'a'
```

`xoutil.objects.`**`copy_class`**(*cls*, *meta=None*, *ignores=None*, *new_attrs=None*, *new_name=None*)

Copies a class definition to a new class.

The returned class will have the same name, bases and module of *cls*.

Parameters

- **`meta`** – If None, the *type(cls)* of the class is used to build the new class, otherwise this must be a *proper* metaclass.
- **`ignores`** – A sequence of attributes names that should not be copied to the new class.

  An item may be callable accepting a single argument *attr* that must return a non-null value if the the *attr* should be ignored.
- **`new_attrs`** (*`dict`*) – New attributes the class must have. These will take precedence over the attributes in the original class.
- **`new_name`** – The name for the copy. If not provided the name will copied.

New in version 1.4.0.

Changed in version 1.7.1: The *ignores* argument must an iterable of strings or callables. Removed the glob-pattern and regular expressions as possible values. They are all possible via the callable variant.

New in version 1.7.1: The *new_name* argument.

`xoutil.objects.`**`fulldir`**(*obj*)

Return a set with all attribute names defined in *obj*

class `xoutil.objects.`**`classproperty`**

A descriptor that behaves like property for instances but for classes.

Example of its use:

```
class Foobar(object):
    @classproperty
    def getx(cls):
        return cls._x
```

A writable *classproperty* is difficult to define, and it's not intended for that case because 'setter', and 'deleter' decorators can't be used for obvious reasons. For example:

```
class Foobar(object):
    x = 1
    def __init__(self, x=2):
        self.x = x
    def _get_name(cls):
        return str(cls.x)
    def _set_name(cls, x):
        cls.x = int(x)
    name = classproperty(_get_name, _set_name)
```

New in version 1.4.1.

Changed in version 1.8.0: Inherits from *property*

`xoutil.objects.`**`get_first_of`**(*sources*, **keys*, *default=None*, *pred=None*)

Return the value of the first occurrence of any of the specified *keys* in *source* that matches *pred* (if given).

Both *source* and *keys* has the same meaning as in *`iterate_over()`*.

Parameters

- **`default`** – A value to be returned if no key is found in *source*.
- **`pred`** – A function that should receive a single value and return False if the value is not acceptable, and thus *get_first_of* should look for another.

Changed in version 1.5.2: Added the *pred* option.

`xoutil.objects.`**`xdir`**(*obj*, *filter=None*, *attr_filter=None*, *value_filter=None*, *getattr=None*)

Return all `(attr, value)` pairs from *obj* that `attr_filter(attr)` and `value_filter(value)` are both True.

Parameters

- **`obj`** – The object to be instrospected.
- **`filter`** – *optional* A filter that will be passed both the attribute name and it's value as two positional arguments. It should return True for attrs that should be yielded.

  Note: If passed, both *attr_filter* and *value_filter* will be ignored.

- **`attr_filter`** – *optional* A filter for attribute names. *Deprecated since 1.4.1*
- **`value_filter`** – *optional* A filter for attribute values. *Deprecated since 1.4.1*
- **`getter`** – *optional* A function with the same signature that `getattr` to be used to get the values from *obj*.

Deprecated since version 1.4.1: The use of params *attr_filter* and *value_filter*.

`xoutil.objects.`**`fdir`**(*obj*, *filter=None*, *attr_filter=None*, *value_filter=None*, *getattr=None*)

Similar to *`xdir()`* but yields only the attributes names.

`xoutil.objects.`**`smart_copy`**(**sources*, *target*, ***, *defaults=False*)

Copies the first apparition of attributes (or keys) from *sources* to *target*.

Parameters

- **`sources`** – The objects from which to extract keys or attributes.
- **`target`** – The object to fill.
- **`defaults`** (*`Either a bool, a dictionary, an iterable or a callable.`*) – Default values for the attributes to be copied as explained below. Defaults to False.

Every *sources* and *target* are always positional arguments. There should be at least one source. *target* will always be the last positional argument.

If *defaults* is a dictionary or an iterable then only the names provided by itering over *defaults* will be copied. If *defaults* is a dictionary, and one of its key is not found in any of the *sources*, then the value of the key in the dictionary is copied to *target* unless:

- It's the value *~xoutil.symbols.Undefined*.
- An exception object
- A sequence with is first value being a subclass of Exception. In which case `adapt_exception` is used.

In these cases a KeyError is raised if the key is not found in the sources.

If *default* is an iterable and a key is not found in any of the sources, None is copied to *target*.

If *defaults* is a callable then it should receive one positional arguments for the current *attribute name* and several keyword arguments (we pass `source`) and return either True or False if the attribute should be copied.

If *defaults* is False (or None) only the attributes that do not start with a "_" are copied, if it's True all attributes are copied.

When *target* is not a mapping only valid Python identifiers will be copied.

Each *source* is considered a mapping if it's an instance of *collections.Mapping* or a *MappingProxyType*.

The *target* is considered a mapping if it's an instance of *collections.MutableMapping*.

Returns *target*.

Changed in version 1.7.0: *defaults* is now keyword only.

`xoutil.objects.`**`extract_attrs`**(*obj*, **names*, *default=Unset*)

Extracts all *names* from an object.

If *obj* is a Mapping, the names will be search in the keys of the *obj*; otherwise the names are considered regular attribute names.

If *default* is Unset and any name is not found, an AttributeError is raised, otherwise the *default* is used instead.

Returns a tuple if there are more that one name, otherwise returns a single value.

New in version 1.4.0.

Changed in version 1.5.3: Each *name* may be a path like in `get_traverser()`, but only "." is allowed as separator.

`xoutil.objects.`**`traverse`**(*obj*, *path*, *default=Unset*, *sep='.'*, *getter=None*)

Traverses an object's hierarchy by performing an attribute get at each level.

This helps getting an attribute that is buried down several levels deep. For example:

```
traverse(request, 'session.somevalue')
```

If *default* is not provided (i.e is `Unset`) and any component in the path is not found an AttributeError exceptions is raised.

You may provide *sep* to change the default separator.

You may provide a custom *getter*. By default, does an `smart_getter()` over the objects. If provided *getter* should have the signature of `getattr()`.

See `get_traverser()` if you need to apply the same path(s) to several objects. Actually this is equivalent to:

```
get_traverser(path, default=default, sep=sep, getter=getter)(obj)
```

`xoutil.objects.`**`get_traverser`**`(`**paths*, *default=Unset*, *sep='.'*, *getter=None*`)`

Combines the power of `traverse()` with the expectations from both `operator.itemgetter()` and `operator.attrgetter()`.

Parameters **`paths`** – Several paths to extract.

Keyword arguments has the same meaning as in `traverse()`.

Returns A function the when invoked with an *object* traverse the object finding each *path*.

New in version 1.5.3.

`xoutil.objects.`**`dict_merge`**`(`**dicts*, ***other*`)`

Merges several dicts into a single one.

Merging is similar to updating a dict, but if values are non-scalars they are also merged is this way:

- Any two `sequences` or `sets` are joined together.
- Any two mappings are recursively merged.
- Other types are just replaced like in `update()`.

If for a single key two values of incompatible types are found, raise a TypeError. If the values for a single key are compatible but different (i.e a list an a tuple) the resultant type will be the type of the first apparition of the key, unless for mappings which are always cast to dicts.

No matter the types of *dicts* the result is always a dict.

Without arguments, return the empty dict.

`xoutil.objects.`**`smart_getattr`**`(`*name*, **sources*, ***kwargs*`)`

Gets an attr by *name* for the first source that has it.

This is roughly that same as:

```
get_first_of(sources, name, default=Unset, **kwargs)
```

Warning: Deprecated since 1.5.1

`xoutil.objects.`**`pop_first_of`**`(`*source*, **keys*, *default=None*`)`

Similar to `get_first_of()` using as *source* either an object or a mapping and deleting the first attribute or key.

Examples:

```
>>> somedict = dict(bar='bar-dict', eggs='eggs-dict')

>>> class Foo(object): pass
>>> foo = Foo()
>>> foo.bar = 'bar-obj'
>>> foo.eggs = 'eggs-obj'

>>> pop_first_of((somedict, foo), 'eggs')
'eggs-dict'

>>> pop_first_of((somedict, foo), 'eggs')
'eggs-obj'

>>> pop_first_of((somedict, foo), 'eggs') is None
True

>>> pop_first_of((foo, somedict), 'bar')
'bar-obj'

>>> pop_first_of((foo, somedict), 'bar')
'bar-dict'

>>> pop_first_of((foo, somedict), 'bar') is None
True
```

`xoutil.objects.`**`get_and_del_attr`**`(`*obj*, *name*, *default=None*`)`
Deprecated alias for `popattr()`.

`xoutil.objects.`**`get_and_del_first_of`**`(`*source*, **keys*, *default=None*`)`
Deprecated alias for `pop_first_of()`.

class `xoutil.objects.`**`metaclass`**`(`*meta*, ***kwargs*`)`
Deprecated alias of `xoutil.eight.meta.metaclass`.

New in version 1.4.1.

Changed in version 1.7.0: Deprecated in favor of `xoutil.eight.meta.metaclass()`.

`xoutil.objects.`**`fix_method_documentation`**`(`*cls*, *method_name*, *ignore=None*, *min_length=10*, *deep=1*, *default=None*`)`
Fix the documentation for the given class using its super-classes.

This function may be useful for shells or Python Command Line Interfaces (CLI).

If *cls* has an invalid documentation, super-classes are recursed in MRO until a documentation definition was made at any level.

Parameters

- **`ignore`** – could be used to specify which classes to ignore by specifying its name in this list.
- **`min_length`** – specify that documentations with less that a number of characters, also are ignored.

`xoutil.objects.`**`multi_getter`**`(`*source*, **ids*`)`
Get values from *source* of all given *ids*.

Parameters

- **`source`** – Any object but dealing with differences between mappings and other object types.

- **ids** – Identifiers to get values from *source*.

  An ID item could be:

  – a string: is considered a key, if *source* is a mapping, or an attribute name if *source* is an instance of any other type.

  – a collection of strings: find the first valid value in *source* evaluating each item in this collection using the above logic.

Example:

```
>>> d = {'x': 1, 'y': 2, 'z': 3}
>>> list(multi_getter(d, 'a', ('y', 'x'), ('x', 'y'), ('a', 'z', 'x')))
[None, 2, 1, 3]

>>> next(multi_getter(d, ('y', 'x'), ('x', 'y')), '---')
2

>>> next(multi_getter(d, 'a', ('b', 'c'), ('e', 'f')), '---') is None
True
```

New in version 1.7.1.

`xoutil.objects.`**`get_branch_subclasses`**(*cls*)

Similar to `type.__subclasses__()` but recursive.

Only return sub-classes in branches (those with no sub-classes). Instead of returning a list, yield each valid value.

New in version 1.7.0.

`xoutil.params` – Tools for managing function arguments

Tools for managing function arguments.

New in version 1.7.1.

`xoutil.params.`**`MAX_ARG_COUNT`** **= 1048576**

The maximum number of positional arguments allowed when calling a function.

class `xoutil.params.`**`ParamManager`**(*args*, *kwds*)

Function parameters parser.

For example:

```
def wraps(*args, **kwargs):
    pm = ParamManager(args, kwargs)
    name = pm(0, 1, 'name', coerce=str)
    wrapped = pm(0, 1, 'wrapped', coerce=valid(callable))
    ...
```

See *`ParamSchemeRow`* and *`ParamScheme`* classes to pre-define and validate schemes for extracting parameter values in a consistent way.

New in version 1.8.0.

`remainder`()

Return not consumed values in a mapping.

class `xoutil.params.`**`ParamScheme`**`(*rows)`

Full scheme for a `ParamManager` instance call.

This class receives a set of `ParamSchemeRow` instances and validate them as a whole.

New in version 1.8.0.

`defaults`

Return a mapping with all valid default values.

`items``()`

Partial compatibility with mappings.

`keys``()`

Partial compatibility with mappings.

class `xoutil.params.`**`ParamSchemeRow`**`(*ids, **options)`

Scheme row for a `ParamManager` instance call.

This class validates identifiers and options at this level; these checks are not done in a call to get a parameter value.

Normally this class is used as part of a full `ParamScheme` composition.

Additionally to the options can be passed to `ParamManager.__call__()`‘, this class can be instanced with:

- ‘key’: an identifier to be used when the parameter is only positional or when none of the possible keyword aliases must be used as the primary-key.

New in version 1.8.0.

`default`

Returned value if parameter value is absent.

If not defined, special value `none` is returned.

`key`

The primary key for this scheme-row definition.

This concept is a little tricky (the first string identifier if some is given, if not then the first integer). This definition is useful, for example, to return remainder not consumed values after a scheme process is completed (see `ParamManager.remainder()` for more information).

`xoutil.params.`**`check_count`**`(args, low, high=1048576, caller=None)`

Check the positional arguments actual count against constrains.

Parameters

- **`args`** – The args to check count, normally is a tuple, but an integer is directly accepted.
- **`low`** – Integer expressing the minimum count allowed.
- **`high`** – Integer expressing the maximum count allowed.
- **`caller`** – Name of the function issuing the check, its value is used only for error reporting.

New in version 1.8.0.

`xoutil.params.`**`check_default`**`(absent=Undefined)`

Get a default value passed as a last excess positional argument.

Parameters **`absent`** – The value to be used by default if no one is given. Defaults to `Undefined`.

For example:

```
def get(self, name, *default):
    from xoutil.params import check_default, Undefined
    if name in self.inner_data:
        return self.inner_data[name]
    elif check_default()(*default) is not Undefined:
        return default[0]
    else:
        raise KeyError(name)
```

New in version 1.8.0.

`xoutil.params.`**`issue_9137`**`(`*args*`)`

Parse arguments for methods, fixing issue 9137 (self ambiguity).

There are methods that expect 'self' as valid keyword argument, this is not possible if this name is used explicitly:

```
def update(self, *args, **kwds):
    ...
```

To solve this, declare the arguments as `method_name(*args, **kwds)`, and in the function code:

```
self, args = issue_9137(args)
```

Returns (self, remainder positional arguments in a tuple)

New in version 1.8.0.

`xoutil.params.`**`keywords_only`**`(`*func*`)`

Make a function to accepts its keywords arguments as keywords-only.

In Python 3 parlance this would make:

```
func(a, b=None)
```

become:

```
func(a, *, b=None).
```

In Python 3 this decorator does nothing. If *func* does not have any keyword arguments, return *func*.

There's a pathological case when you define:

```
func(a, b=None, *args)
```

In such a case if you call `func(1, 2, b=3)` we can't actually call the original function with `a=1`, `args=(2, )` and `b=3`. This case also raises a TypeError.

New in version 1.8.0.

`xoutil.params.`**`pop_keyword_arg`**`(`*kwargs*, *names*, *default=Undefined*`)`

Return the value of a keyword argument.

Parameters

- **`kwargs`** – The mapping with passed keyword arguments.
- **`names`** – Could be a single name, or a collection of names.
- **`default`** – The default value to return if no value is found.

New in version 1.8.0.

`xoutil.params.`**`single`**(*args*, *kwds*)

Return a true value only when a unique argument is given.

Wnen needed, the most suitable result will be wrapped using the *`Maybe`*.

New in version 1.8.0.

Because the nature of this tool, the term "parameter" will be used in this documentation to reference those of the represented client function, and the term "argument" for referencing those pertaining to *`ParamManager`* methods.

class `xoutil.params.`**`ParamManager`**(*args*, *kwds*)

Function parameters parser.

For example:

```
def wraps(*args, **kwargs):
    pm = ParamManager(args, kwargs)
    name = pm(0, 1, 'name', coerce=str)
    wrapped = pm(0, 1, 'wrapped', coerce=valid(callable))
    ...
```

See *`ParamSchemeRow`* and *`ParamScheme`* classes to pre-define and validate schemes for extracting parameter values in a consistent way.

New in version 1.8.0.

When use this class as a callable, each identifier could be an integer or a string, respectively representing indexes in the positional and names in the keyword parameters. Negative indexes are treated as in Python tuples or lists.

Several identifiers must be unambiguous, or because some integers are already marked as consumed in previous calls or because an option coerce function validate only one position. In the case of names, when one value is hit, all remainder names must be absent in the *kwargs* parameters.

'coerce' option could be a callable, or a Python type (or a tuple of types). When callable, must return a coerced valid value or *Invalid*; when type or types, *isinstance* standard function is used to check.

`__call__`(**ids*, ***options*)

Get a parameter value.

Parameters

- **`ids`** – parameter identifiers.
- **`options`** – keyword argument options.

Options could be:

- 'default': value used if the parameter is absent;
- 'coerce': check if a value is valid or not and convert to its definitive value; see *`xoutil.values`* module for more information.

`__init__`(*args*, *kwds*)

Created with actual parameters of a client function.

`remainder`()

Return not consumed values in a mapping.

class `xoutil.params.`**`ParamScheme`**(**rows*)

Full scheme for a *`ParamManager`* instance call.

This class receives a set of *`ParamSchemeRow`* instances and validate them as a whole.

New in version 1.8.0.

__call__(*args*, *kwds*, *strict=False*)
Get a mapping with all resulting values.

If special value 'none' is used as 'default' option in a scheme-row, corresponding value isn't returned in the mapping if the parameter value is missing.

__getitem__(*idx*)
Obtain the scheme-row by a given index.

__iter__()
Iterate over all defined scheme-rows.

__len__()
The defined scheme-rows number.

defaults
Return a mapping with all valid default values.

class `xoutil.params.`**ParamSchemeRow**(**ids*, ***options*)
Scheme row for a `ParamManager` instance call.

This class validates identifiers and options at this level; these checks are not done in a call to get a parameter value.

Normally this class is used as part of a full `ParamScheme` composition.

Additionally to the options can be passed to `ParamManager.__call__()`', this class can be instanced with:

- 'key': an identifier to be used when the parameter is only positional or when none of the possible keyword aliases must be used as the primary-key.

New in version 1.8.0.

This class generates callable instances receiving one `ParamManager` instance as its single argument.

__call__(**args*, ***kwds*)
Execute a scheme-row using as argument a *ParamManager* instance.

The concept of `ParamManager` instance argument is a little tricky: when a variable number of arguments is used, if only one positional and is already an instance of `ParamManager`, it is directly used; if two, the first is a *tuple* and the second is a *dict*, these are considered the constructor arguments of the new instance; otherwise all arguments are used to build the new instance.

default
Returned value if parameter value is absent.

If not defined, special value `none` is returned.

key
The primary key for this scheme-row definition.

This concept is a little tricky (the first string identifier if some is given, if not then the first integer). This definition is useful, for example, to return remainder not consumed values after a scheme process is completed (see `ParamManager.remainder()` for more information).

`xoutil.progress` - Console progress utils

Tool to show a progress percent in the terminal.

class `xoutil.progress.`**`Progress`**(*max_value=100*, *delta=1*, *first_message=None*, *display_width=None*)

Print a progress percent to the console. Also the elapsed and the estimated times.

To signal an increment in progress just call the instance and (optionally) pass a message like in:

```
progress = Progress(10)
for i in range(10):
    progress()
```

`xoutil.records` - Records definitions

Records definitions.

A record allows to describe plain external data and a simplified model to *read* it. The main use of records is to represent data that is read from a CSV file.

See the *record* class to find out how to use it.

class `xoutil.records.`**`record`**(*raw_data*)

Base record class.

Records allow to represent a sequence or mapping of values extracted from external sources into a dict-like Python value.

The first use-case for this abstraction is importing data from a CSV file. You could represent each line as an instance of a properly defined record.

An instance of a record would represent a single *line* (or row) from the external data source.

Records are expected to declare *fields*. Each field must be a CAPITALIZED valid identifier like:

```
>>> class INVOICE(record):
...     ID = 0
...     REFERENCE = 1
```

Fields must be integers or plain strings. Fields must not begin with an underscore ("_"). External data lines are required to support indexes of those types.

You could use either the classmethod `get_field()` to get the value of field in a single line (data as provided by the external source):

```
>>> line = (1, 'AA20X138874Z012')
>>> INVOICE.get_field(line, INVOICE.REFERENCE)
'AA20X138874Z012'
```

You may also have an instance:

```
>>> invoice = INVOICE(line)
>>> invoice.reference
'AA20X138874Z012'
```

Note: Instances attributes are renamed to lowercase. So you **must** not create any other attribute that has the same name as a field in lowercase, or else it will be overwritten.

You could define *readers* for any field. For instance if you have a “CREATED_DATETIME” field you may create a “_created_datetime_reader” function that will be used to parse the raw value of the instance into an expected type. See the *included readers builders below*.

Readers are always cast as *staticmethods*, whether or not you have explicitly stated that fact:

```
>>> from dateutil import parser
>>> class BETTER_INVOICE(INVOICE):
...     CREATED_TIME = 2
...     _created_time_reader = lambda val: parser.parse(val)
...

>>> line = (1, 'AA20X138874Z012', '2014-02-17T17:29:21.965053')
>>> BETTER_INVOICE.get_field(line, BETTER_INVOICE.CREATED_TIME)
datetime.datetime(2014, 2, 17, 17, 29, 21, 965053)
```

Warning: Creating readers for fields defined in super classes is not directly supported. To do so, you **must** declare the reader as a staticmethod yourself.

Note: Currently there’s no concept of relationship between rows in this model. We are evaluating whether by placing a some sort of context into the *kwargs* argument would be possible to write readers that fetch other instances.

Included reader builders

The following functions *build* readers for standards types.

Note: You cannot use these functions themselves as readers, but you must call them to obtain the desired reader.

All these functions have a pair of keywords arguments *nullable* and *default*. The argument *nullable* indicates whether the value must be present or not. The function `check_nullable()` implements this check and allows other to create their own builders with the same semantic.

`xoutil.records.`**`datetime_reader`**(*format*, *nullable=False*, *default=None*, *strict=True*)

Returns a datetime reader.

Parameters

- **`format`** – The format the datetime is expected to be in the external data. This is passed to `datetime.datetime.strptime()`.
- **`strict`** – Whether to be strict about datetime format.

The reader works first by passing the value to strict `datetime.datetime.strptime()` function. If that fails with a ValueError and strict is True the reader fails entirely.

If strict is False, the worker applies different rules. First if the *dateutil* package is installed its parser module is tried. If *dateutil* is not available and nullable is True, return None; if nullable is False and default is not null (as in `isnull()`), return *default*, otherwise raise a ValueError.

`xoutil.records.`**`boolean_reader`**(*true=(‘1’,)*, *nullable=False*, *default=None*)

Returns a boolean reader.

Parameters **true** – A collection of raw values considered to be True. Only the values in this collection will be considered True values.

`xoutil.records.`**`integer_reader`**(*nullable=False*, *default=None*)
Returns an integer reader.

`xoutil.records.`**`decimal_reader`**(*nullable=False*, *default=None*)
Returns a Decimal reader.

`xoutil.records.`**`float_reader`**(*nullable=False*, *default=None*)
Returns a float reader.

`xoutil.records.`**`date_reader`**(*format*, *nullable=False*, *default=None*, *strict=True*)
Return a date reader.

This is similar to `datetime_reader()` but instead of returning a `datetime.datetime` it returns a *datetime.date*.

Actually this function delegates to `datetime_reader()` most of its functionality.

Checking for null values

`xoutil.records.`**`isnull`**(*val*)
Return True if *val* is null.

Null values are None, the empty string and any False instance of `xoutil.symbols.boolean`.

Notice that 0, the empty list and other false values in Python are not considered null. This allows that the CSV null (the empty string) is correctly treated while other sources that provide numbers (and 0 is a valid number) are not misinterpreted as null.

`xoutil.records.`**`check_nullable`**(*val*, *nullable*)
Check the restriction of nullable.

Return True if the val is non-null. If nullable is True and the val is null returns False. If *nullable* is False and *val* is null, raise a ValueError.

Test for null is done with function `isnull()`.

These couple of functions allows you to define new builders that use the same null concept. For instance, if you need readers that parse dates in diferent locales you may do:

```
def date_reader(nullable=False, default=None, locale=None):
    from xoutil.records import check_nullable
    from babel.dates import parse_date, LC_TIME
    from datetime import datetime
    if not locale:
        locale = LC_TIME

    def reader(value):
        if check_nullable(value, nullable):
            return parse_date(value, locale=locale)
        else:
            return default
    return reader
```

`xoutil.string` - Common string operations

Some additions for *string* standard module.

In this module *str* and *unicode* types are not used because Python 2 and Python 3 treats strings differently (see *String Ambiguity in Python* for more information). The types *bytes* and *text_type* will be used instead with the following conventions:

- In Python 2 *str* is synonym of *bytes* and both (*unicode* and 'str') are both string types inheriting form *basestring*.
- In Python 3 *str* is always unicode but *unicode* and *basestring* types doesn't exists. *bytes* type can be used as an array of one byte each item.

`xoutil.string.`**`cut_any_prefix`**`(`*value*, **prefixes*`)`
Apply `cut_prefix()` for the first matching prefix.

`xoutil.string.`**`cut_any_suffix`**`(`*value*, **suffixes*`)`
Apply `cut_suffix()` for the first matching suffix.

`xoutil.string.`**`cut_prefix`**`(`*value*, *prefix*`)`
Removes the leading *prefix* if exists, else return *value* unchanged.

`xoutil.string.`**`cut_prefixes`**`(`*value*, **prefixes*`)`
Apply `cut_prefix()` for all provided prefixes in order.

`xoutil.string.`**`cut_suffix`**`(`*value*, *suffix*`)`
Removes the tailing *suffix* if exists, else return *value* unchanged.

`xoutil.string.`**`cut_suffixes`**`(`*value*, **suffixes*`)`
Apply `cut_suffix()` for all provided suffixes in order.

`xoutil.string.`**`error2str`**`(`*error*`)`
Convert an error to string.

`xoutil.string.`**`make_a10z`**`(`*string*`)`
Utility to find out that "internationalization" is "i18n".

Examples:

```
>>> print(make_a10z('parametrization'))
p13n
```

`xoutil.string.`**`normalize_slug`**`(`**args*, ***kw*`)`
Return the normal-form of a given string value that is valid for slugs.

Convert all non-ascii to valid characters, whenever possible, using unicode 'NFKC' normalization and lower-case the result. Replace unwanted characters by the value of *replacement* (remove extra when repeated).

Default valid characters are `[_a-z0-9]`. Extra arguments *invalid_chars* and *valid_chars* can modify this standard behaviour, see next:

Parameters

- **`value`** – The source value to slugify.
- **`replacement`** – A character to be used as replacement for unwanted characters. Could be both, the first extra positional argument, or as a keyword argument. Default value is a hyphen ('-').

  There will be a contradiction if this argument contains any invalid character (see *invalid_chars*). `None`, or `False`, will be converted converted to an empty string for backward compatibility with old versions of this function, but not use this, will be deprecated.

- **invalid_chars** – Characters to be considered invalid. There is a default set of valid characters which are kept in the resulting slug. Characters given in this parameter are removed from the resulting valid character set (see *valid_chars*).

  Extra argument values can be used for compatibility with *invalid_underscore* argument in deprecated *normalize_slug* function:

  – `True` is a synonymous of underscore `"_"`.

  – `False` or `None`: An empty set.

  Could be given as a name argument or in the second extra positional argument. Default value is an empty set.

- **valid_chars** – A collection of extra valid characters. Could be either a valid string, any iterator of strings, or `None` to use only default valid characters. Non-ASCII characters are ignored.

Examples:

```
>>> slugify('  Á.e i  Ó  u  ') == 'a-e-i-o-u'
True

>>> slugify(' Á.e i  Ó  u  ', '.', invalid_chars='AU') == 'e.i.o'
True

>>> slugify('  Á.e i  Ó  u  ', valid_chars='.') == 'a.e-i-o-u'
True

>>> slugify('_x', '_') == '_x'
True

>>> slugify('-x', '_') == 'x'
True

>>> slugify(None) == 'none'
True

>>> slugify(1 == 1)  == 'true'
True

>>> slugify(1.0) == '1-0'
True

>>> slugify(135) == '135'
True

>>> slugify(123456, '', invalid_chars='52') == '1346'
True

>>> slugify('_x', '_') == '_x'
True
```

Changed in version 1.5.5: Added the *invalid_underscore* parameter.

Changed in version 1.6.6: Replaced the *invalid_underscore* paremeter by *invalids*. Added the *valids* parameter.

Changed in version 1.7.2: Clarified the role of *invalids* with regards to *replacement*.

Changed in version 1.8.0: Deprecate the *invalids* paremeter name in favor of *invalid_chars*, also deprecate the *valids* paremeter name in favor of *valid_chars*.

`xoutil.string.slugify`(*value*, **args*, ***kwds*)

Return the normal-form of a given string value that is valid for slugs.

Convert all non-ascii to valid characters, whenever possible, using unicode 'NFKC' normalization and lower-case the result. Replace unwanted characters by the value of *replacement* (remove extra when repeated).

Default valid characters are `[_a-z0-9]`. Extra arguments *invalid_chars* and *valid_chars* can modify this standard behaviour, see next:

Parameters

- **value** – The source value to slugify.
- **replacement** – A character to be used as replacement for unwanted characters. Could be both, the first extra positional argument, or as a keyword argument. Default value is a hyphen ('-').

  There will be a contradiction if this argument contains any invalid character (see *invalid_chars*). `None`, or `False`, will be converted converted to an empty string for backward compatibility with old versions of this function, but not use this, will be deprecated.
- **invalid_chars** – Characters to be considered invalid. There is a default set of valid characters which are kept in the resulting slug. Characters given in this parameter are removed from the resulting valid character set (see *valid_chars*).

  Extra argument values can be used for compatibility with *invalid_underscore* argument in deprecated *normalize_slug* function:

  – `True` is a synonymous of underscore `"_"`.

  – `False` or `None`: An empty set.

  Could be given as a name argument or in the second extra positional argument. Default value is an empty set.
- **valid_chars** – A collection of extra valid characters. Could be either a valid string, any iterator of strings, or `None` to use only default valid characters. Non-ASCII characters are ignored.

Examples:

```
>>> slugify('  Á.e i  Ó  u  ') == 'a-e-i-o-u'
True

>>> slugify(' Á.e i  Ó  u  ', '.', invalid_chars='AU') == 'e.i.o'
True

>>> slugify('  Á.e i  Ó  u  ', valid_chars='.') == 'a.e-i-o-u'
True

>>> slugify('_x', '_') == '_x'
True

>>> slugify('-x', '_') == 'x'
True

>>> slugify(None) == 'none'
True

>>> slugify(1 == 1)  == 'true'
True
```

```
>>> slugify(1.0) == '1-0'
True

>>> slugify(135) == '135'
True

>>> slugify(123456, '', invalid_chars='52') == '1346'
True

>>> slugify('_x', '_') == '_x'
True
```

Changed in version 1.5.5: Added the *invalid_underscore* parameter.

Changed in version 1.6.6: Replaced the *invalid_underscore* paremeter by *invalids*. Added the *valids* parameter.

Changed in version 1.7.2: Clarified the role of *invalids* with regards to *replacement*.

Changed in version 1.8.0: Deprecate the *invalids* paremeter name in favor of *invalid_chars*, also deprecate the *valids* paremeter name in favor of *valid_chars*.

`xoutil.symbols` – Basic function argument manager

Special logical values like Unset, Undefined, Ignored, Invalid, ...

All values only could be *True* or *False* but are intended in places where *None* is expected to be a valid value or for special Boolean formats.

`xoutil.symbols.`**`Ignored`** **= Ignored**
To be used in arguments that are currently ignored because they are being deprecated. The only valid reason to use *Ignored* is to signal ignored arguments in method's/function's signature

`xoutil.symbols.`**`Invalid`** **= Invalid**
To be used in functions resulting in a fail where False could be a valid value.

class `xoutil.symbols.`**`MetaSymbol`**
Meta-class for symbol types.

`nameof`(*s*)
Get the name of a symbol instance (*s*).

`parse`(*name*)
Returns instance from a string.

Standard Python Boolean values are parsed too.

`xoutil.symbols.`**`This`** **= This**
To be used as a mark for current context as a mechanism of comfort.

`xoutil.symbols.`**`Undefined`** **= Undefined**
False value for local scope use or where `Unset` could be a valid value

`xoutil.symbols.`**`Unset`** **= Unset**
False value, mainly for function parameter definitions, where None could be a valid value.

class `xoutil.symbols.`**`boolean`**(**args*, ***kwds*)
Instances are custom logical values (*True* or *False*).

See `__getitem__()` operator for information on constructor arguments.

For example:

```
>>> true = boolean('true', True)
>>> false = boolean('false')
>>> none = boolean('false')
>>> unset = boolean('unset')

>>> class X(object):
...      attr = None

>>> getattr(X(), 'attr') is not None
False

>>> getattr(X(), 'attr', false) is not false
True

>>> none is false
True

>>> false == False
True

>>> false == unset
True

>>> false is unset
False

>>> true == True
True
```

class `xoutil.symbols.`**`symbol`**`(`**args*, ***kwds*`)`

Instances are custom symbols.

See `__getitem__()` operator for information on constructor arguments.

For example:

```
>>> ONE2MANY = symbol('ONE2MANY')
>>> ONE_TO_MANY = symbol('ONE2MANY')

>>> ONE_TO_MANY ONE2MANY
True
```

`xoutil.validators` – value validators

Some generic value validators and regular expressions and validation functions for several identifiers.

`xoutil.validators.`**`check`**`(`*value*, *validator*, *msg=None*`)`

Check a *value* with a *validator*.

Argument *validator* could be a callable, a type, or a tuple of types.

Return True if the value is valid.

Examples:

```
>>> check(1, int)
True
```

```
>>> check(10, lambda x: x <= 100, 'must be less than or equal to 100')
True

>>> check(11/2, (int, float))
True
```

`xoutil.validators.`**`check_no_extra_kwargs`**`(`*kwargs*`)`

Check that no extra keyword arguments are still not processed.

For example:

```
>>> from xoutil.validators import check_no_extra_kwargs
>>> def only_safe_arg(**kwargs):
...     safe = kwargs.pop('safe', False)
...     check_no_extra_kwargs(kwargs)
...     print('OK for safe:', safe)
```

`xoutil.validators.`**`is_type`**`(`*cls*`)`

Return a validator with the same name as the type given as argument *value*.

Parameters **`cls`** – Class or type or tuple of several types.

`xoutil.validators.`**`ok`**`(`*value*, **checkers*, ***kwargs*`)`

Validate a value with several checkers.

Return the value if it is Ok, or raises an *ValueError* exception if not.

Arguments:

Parameters

- **`value`** – the value to validate
- **`checkers`** – a variable number of checkers (at least one), each one could be a type, a tuple of types of a callable that receives the value and returns if the value is valid or not. In order the value is considered valid, all checkers must validate the value.
- **`message`** – keyword argument to be used in case of error; will be the argument of *ValueError* exception; could contain the placeholders `{value}` and `{type}`; a default value is used if this argument is not given.
- **`msg`** – an alias for "message"
- **`extra_checkers`** – In order to create validators using *partial*. Must be a tuple.

Keyword arguments are not validated to be correct.

This function could be used with type-definitions for arguments, see `xoutil.fp.prove.semantic.TypeCheck`.

Examples:

```
>>> ok(1, int)
1

>>> ok(10, int, lambda x: x < 100, message='Must be integer under 100')
10

>>> ok(11/2, (int, float))
5.5
```

```
>>> ok(11/2, int, float)
5.5

>>> try:
...     res = ok(11/2, int)
... except ValueError:
...     res = '---'
>>> res
'---'
```

`xoutil.validators.`**`predicate`**`(`**checkers*, ***kwargs*`)`

Return a validation checker for types and simple conditions.

Parameters

- **`checkers`** – A variable number of checkers; each one could be:
  - A type, or tuple of types, to test valid values with `isinstance(value, checker)`
  - A set or mapping of valid values, the value is valid if contained in the checker.
  - A tuple of other inner checkers, if any of the checkers validates a value, the value is valid (OR).
  - A list of other inner checkers, all checkers must validate the value (AND).
  - A callable that receives the value and returns True if the value is valid.
  - `True` and `False` could be used as checkers always validating or invalidating the value.

  An empty list or no checker is synonym of `True`, an empty tuple, set or mapping is synonym of `False`.
- **`name`** – Keyword argument to be used in case of error; will be the argument of *ValueError* exception; could contain the placeholders `{value}` and `{type}`; a default value is used if this argument is not given.
- **`force_name`** – Keyword argument to force a name if not given.

In order to obtain good documentations, use proper names for functions and lambda arguments.

With this function could be built real type checkers, for example:

```
>>> is_valid_age = predicate((int, float), lambda age: 0 < age <= 120)
>>> is_valid_age(100)
True

>>> is_valid_age(130)
False

>>> always_true = predicate(True)
>>> always_true(False)
True

>>> always_false = predicate(False)
>>> always_false(True)
False

>>> always_true = predicate()
>>> always_true(1)
True
```

```
>>> always_true('any string')
True

>>> always_false = predicate(())
>>> always_false(1)
False

>>> always_false('any string')
False
```

Contents:

`xoutil.validators.identifiers` – Simple identifiers validators

Regular expressions and validation functions for several identifiers.

`xoutil.validators.identifiers.`**`is_valid_identifier`**`(`*name*`)`
Returns True if *name* a valid Python identifier.

Note: Only Python 2's version of valid identifier. This means that some Python 3 valid identifiers are not considered valid. This helps to keep things working the same in Python 2 and 3.

`xoutil.validators.identifiers.`**`is_valid_full_identifier`**`(`*name*`)`
Returns True if *name* is a valid dotted Python identifier.

See `is_valid_identifier()` for what "validity" means.

`xoutil.validators.identifiers.`**`is_valid_public_identifier`**`(`*name*`)`
Returns True if *name* is a valid Python identifier that is deemed public.

Convention says that any name starting with a "_" is not public.

See `is_valid_identifier()` for what "validity" means.

`xoutil.values` – coercers (or checkers) for value types

Some generic coercers (or checkers) for value types.

This module coercion function are not related in any way to deprecated old python feature, are similar to a combination of object mold/check:

- *Mold* - Fit values to expected conventions.
- *Check* - These functions must return *nil*[1] special value to specify that expected fit is not possible.

A custom coercer could be created with closures, for an example see `create_int_range_coerce()`.

This module uses *Unset* value to define absent -not being specified- arguments.

Also contains sub-modules to obtain, convert and check values of common types.

New in version 1.7.0.

[1] We don't use Python classic *NotImplemented* special value in order to obtain False if the value is not coerced (*nil*).

class `xoutil.values.`**`MetaCoercer`**

Meta-class for `coercer`.

This meta-class allows that several objects are considered valid instances of `coercer`:

- Functions decorated with `coercer` (used with its decorator facet).
- Instances of any sub-class of `custom`.
- Instances of `coercer` itself.

See the class declaration (`coercer`) for more information.

`xoutil.values.`**`callable_coerce`**`(`*arg*`)`

Check if *arg* is a callable object.

class `xoutil.values.`**`coercer`**

Special coercer class.

This class has several facets:

- Pure type-checkers when a type or tuple of types are received as argument. See `istype` for more information.
- Return equivalent coercer from some special values:
  - Any true value -> identity_coerce
  - Any false or empty value -> void_coerce
- A decorator for functions; when a function is given, decorate it to become a coercer. The mark itself is not enough, functions intended to be coercers must fulfills the protocol (not to produce exception and return *nil* on fails). For example:

```
>>> @coercer
... def age_coerce(arg):
...     res = int_coerce(arg)
...     return res if t(res) and 0 < arg <= 120 else nil

# TODO: Change next, don't use isinstance
>>> isinstance(age_coerce, coercer)
True
```

`xoutil.values.`**`coercer_name`**`(`*arg*, *join=None*`)`

Get the name of a coercer.

Parameters

- **`arg`** – Coercer to get the name. Also processes collections (tuple, list, or set) of coercers. Any other value is considered invalid and raises an exception.
- **`join`** – When a collection is used; if this argument is None a collection of names is returned, if not None then is used to join the items in a resulting string.

  For example:

```
>>> coercer_name((int_coerce, float_coerce))
('int', 'float')

>>> coercer_name((int_coerce, float_coerce), join='-')
'int-float'
```

To obtain pretty-print tuples, use something like:

```
>>> coercer_name((int_coerce, float_coerce),
...              join=lambda arg: '(%s)' % ', '.join(arg))
```

This function not only works with coercers, all objects that fulfill needed protocol to get names will also be valid.

class xoutil.values.**combo**(**coercers*)

Represent a zip composition of several inner *coercers*.

An instance of this class is constructed from a sequence of coercers and the its purpose is coerce a sequence of values. Return a sequence[2] where each item contains the i-th element from applying the i-th coercer to the i-th value from argument sequence:

```
coercers -> (coercer-1, coercer-2, ... )
values -> (value-1, value-2, ... )
combo(coercers)(values) -> (coercer-1(value-1), coercer-2(value-2), ...)
```

If any value is coerced invalid, the function returns *nil* and the combo's instance variable *scope* receives the duple `(failed-value, failed-coercer)`.

The returned sequence is truncated in length to the length of the shortest sequence (coercers or arguments).

If no coercer is given, all sequences are coerced as empty.

class xoutil.values.**compose**(**args*, ***kwargs*)

Returns the composition of several inner *coercers*.

`compose(f1, ... fn)` is equivalent to f1(...(fn(arg))...)``.

If no coercer is given return *`identity_coerce()`*.

Could be considered an "AND" operator with some light differences because the nature of coercers: ordering the coercers is important when some can modify (adapt) original values.

If no value results in *coercers*, a default coercer could be given as a keyword argument; *identity_coerce* is assumed if missing.

xoutil.values.**create_int_range_coerce**(*min*, *max*)

Create a coercer to check integers between a range.

xoutil.values.**create_unique_member_coerce**(*coerce*, *container*)

Useful to wrap member coercers when coercing containers.

See *`iterable`* and *`mapping`*.

Resulting coercer check that a member must be unique (not repeated) after it's coerced.

For example:

```
>>> from xoutil.values import (mapping, create_unique_member_coerce,
...                            int_coerce, float_coerce)

>>> sample = {'1': 1, 2.0: '3', 1.0 + 0j: '4.1'}

>>> dc = mapping(int_coerce, float_coerce)
>>> dc(dict(sample))
{1: 1.0, 2: 3.0}

>>> dc = mapping(create_unique_member_coerce(int_coerce), float_coerce)
>>> dc(dict(sample))
nil
```

[2] The returned sequence is of the same type as the argument sequence if possible.

class `xoutil.values.`**`custom`**`(*args, **kwargs)`

Base class for any custom coercer.

The field *inner* stores an internal data used for the custom coercer; could be a callable, an inner coercer, or a tuple of inner checkers if more than one is needed, ...

The field *scope* stores the exit (not regular) condition: the value that fails or -if needed- a tuple with (exit-value, exit-coercer) or (error-value, error). The exit condition is not always a failure, for example in `some` it is the one that is valid among other inner coercers. To understand better this think on (AND, OR) operators a chain of ANDs exits with the first failure and a chains of ORs exits with the first success.

All custom coercers are callable (must redefine `__call__()`) receiving one argument that must be coerced. For example:

```
>>> def foobar(*args):
...     coerce = pargs(int_coerce)
...     return coerce(args)
```

This class has two protected fields (*_str_join* and *_repr_join*) that are used to call `coercer_name()` in `__str__()` and `__repr__()` special methods.

classmethod **`flatten`**`(obj, avoid=Unset)`

Flatten a coercer set.

Parameters **`obj`** – Could be a coercer representing other inner coercers, or a tuple or list containing coercers.

`xoutil.values.`**`file_coerce`**`(arg)`

Check if *arg* is a file-like object.

`xoutil.values.`**`float_coerce`**`(arg)`

Check if *arg* is a valid float.

Other types are checked (string, int, complex).

`xoutil.values.`**`full_identifier_coerce`**`(arg)`

Check if *arg* is a valid dotted Python identifier.

See `identifier_coerce()` for what "validity" means.

`xoutil.values.`**`identifier_coerce`**`(arg)`

Check if *arg* is a valid Python identifier.

Note: Only Python 2's version of valid identifier. This means that some Python 3 valid identifiers are not considered valid. This helps to keep things working the same in Python 2 and 3.

`xoutil.values.`**`identity_coerce`**`(arg)`

Leaves unchanged the passed argument *arg*.

`xoutil.values.`**`int_coerce`**`(arg)`

Check if *arg* is a valid integer.

Other types are checked (string, float, complex).

class `xoutil.values.`**`istype`**`(*args, **kwargs)`

Pure type-checker.

It's constructed from an argument valid for `types_tuple_coerce()` coercer.

For example:

```
>>> int_coerce = istype(int)

>>> int_coerce(1)
1

>>> int_coerce('1')
nil

>>> number_coerce = istype((int, float, complex))

>>> number_coerce(1.25)
1.25

>>> number_coerce('1.25')
nil
```

class `xoutil.values.`**`iterable`**`(`*member_coerce*, *outer_coerce=True*`)`

Create a inner coercer that coerces an *iterable* member a member.

See constructor for more information.

Return a list, or the same type of source iterable argument if possible.

For example:

```
>>> from xoutil.values import (iterable, int_coerce,
...                            create_unique_member_coerce)

>>> sample = {'1', 1, '1.0'}

>>> sc = iterable(int_coerce)
>>> sc(set(sample)) == {1}
True
```

See *`mapping`* for more details of this problem. The equivalent safe example is:

```
>>> member_coerce = create_unique_member_coerce(int_coerce, sample)
>>> sc = iterable(member_coerce)
>>> sc(set(sample))
nil
```

when executed coerces *arg* (an iterable) member a member using *member_coercer*. If any member coercion fails, the full execution also fails.

There are three types of results when an instance is executed: (1) iterables that are coerced without modifications, (2) the modified ones but conserving its type, and (3) those that are returned in a list.

class `xoutil.values.`**`logical`**`(`**args*, ***kwds*`)`

Represent Common Lisp two special values *t* and *nil*.

Include redefinition of `__call__()` to check values with special semantic:

- When called as `t(arg)`, check if *arg* is not *nil* returning a logical true: the same argument if *arg* is nil or a true boolean value, else return *t*. That means that *False* or *0* are valid true values for Common Lisp but not for Python.
- When called as `nil(arg)`, check if *arg* is *nil* returning *t* or *nil* if not.

Constructor could receive a valid name ('nil' or 't') or any other `boolean` instance.

class `xoutil.values.`**`mapping`**`(*args, **kwargs)`

Create a coercer to check dictionaries.

Receives two coercers, one for keys and one for values.

For example:

```
>>> from xoutil.values import (mapping, int_coerce, float_coerce,
...                             create_unique_member_coerce)

>>> sample = {'1': 1, 2.0: '3', 1.0 + 0j: '4.1'}

>>> dc = mapping(int_coerce, float_coerce)
>>> dc(dict(sample)) == {1: 1.0, 2: 3.0}
True
```

When coercing containers it's probable that members become repeated after coercing them. This could be not desirable (mainly in sets and dictionaries). In those cases use *`create_unique_member_coerce()`* to wrap member coercer. For example:

```
>>> key_coerce = create_unique_member_coerce(int_coerce, sample)
>>> dc = mapping(key_coerce, float_coerce)
>>> dc(dict(sample))
nil
```

Above problem is because it's the same integer (same hash) coerced versions of `'1'` and `1.0+0j`.

This problem of objects of different types that have the same hash is a problem to use a example as below:

```
>>> {1: int, 1.0: float, 1+0j: complex} == {1: complex}
True
```

`xoutil.values.`**`names_coerce`**`(arg)`

Check *arg* as a tuple of valid object names (identifiers).

If only one string is given, is returned as the only member of the resulting tuple.

`xoutil.values.`**`number_coerce`**`(arg)`

Check if *arg* is a valid number (integer or float).

Types that are checked (string, int, float, complex).

class `xoutil.values.`**`pargs`**`(arg_coerce)`

Create a inner coercer that check variable argument passing.

Created coercer closure must always receives an argument that is an valid iterable with all members coerced properly with the argument of this outer creator function.

If the inner closure argument has only a member and this one is not properly coerced but it's an iterabled with all members that coerced well, this member will be the assumed iterable instead the original argument.

In the following example:

```
>>> from xoutil.values import (iterable, int_coerce)

>>> def foobar(*args):
...     coerce = iterable(int_coerce)
...     return coerce(args)

>>> args = (1, 2.0, '3.0')
>>> foobar(*args)
```

```
(1, 2, 3)

>>> foobar(args)
nil
```

An example using *pargs*

```
>>> from xoutil.values import (pargs, int_coerce)

>>> def foobar(*args):
...     # Below, "coercer" receives the returned "inner"
...     coerce = pargs(int_coerce)
...     return coerce(args)

>>> args = (1, 2.0, '3.0')
>>> foobar(*args)
(1, 2, 3)

>>> foobar(args)
(1, 2, 3)
```

The second form is an example of the real utility of this coercer closure: if by error a sequence is passed as it to a function that expect a variable number of argument, this coercer fixes it.

Instance variable *scope* stores the last processed invalid argument.

When executed, usually *arg* is a tuple received by a function as `*args` form.

When executed, returns a tuple, or the same type of source iterable argument if possible.

See *xoutil.params* for a more specialized and full function arguments conformer.

See *combo* for a combined coercer that validate each member with a separate member coercer.

`xoutil.values.`**`positive_int_coerce`**`(`*arg*`)`
Check if *arg* is a valid positive integer.

class `xoutil.values.`**`safe`**`(`*func*`)`
Uses a function (or callable) in a safe way.

Receives a coercer that expects only one argument and returns another value.

If the returned value is a `boolean` (maybe the coercer is a predicate), it's converted to a `logical` instance.

The wrapped coercer is called in a safe way (inside try/except); if an exception is raised the coercer returns `nil` and the error is saved in the instance attribute `scope`.

`xoutil.values.`**`sized_coerce`**`(`*arg*`)`
Return a valid sized iterable from *arg*.

If *arg* is iterable but not sized, is converted to a list. For example:

```
>>> sized_coerce(i for i in range(1, 10, 2))
[1, 3, 5, 7, 9]

>>> s = {1, 2, 3}
>>> sized_coerce(s) is s
True
```

class `xoutil.values.`**`some`**`(`**args*, ***kwargs*`)`
Represent OR composition of several inner *coercers*.

`compose(f1, ... fn)` is equivalent to f1(arg) or f2(arg) ... fn(arg)‘‘ in the sense “the first not *nil*”.

If no coercer is given return *void_coerce()*.

`xoutil.values.`**`type_coerce`**`(`*arg*`)`
Check if *arg* is a valid type.

class `xoutil.values.`**`typecast`**`(`**args*, ***kwargs*`)`
A type-caster.

It’s constructed from an argument valid for *types_tuple_coerce()* coercer. Similar to *istype* but try to convert the value if needed.

For example:

```
>>> int_cast = typecast(int)

>>> int_cast('1')
1

>>> int_cast('1x')
nil
```

`xoutil.values.`**`types_tuple_coerce`**`(`*arg*`)`
Check if *arg* is valid for *isinstance* or *issubclass* 2nd argument.

Type checkers are any class, a type or tuple of types. For example:

```
>>> types_tuple_coerce(object) == (object,)
True

>>> types_tuple_coerce((int, float)) == (int, float)
true

>>> types_tuple_coerce('not-a-type') is nil
True
```

See *type_coerce* for more information.

`xoutil.values.`**`void_coerce`**`(`*arg*`)`
Always *nil*.

Contents:

`xoutil.values.ids` – unique identifiers at different contexts

Utilities to obtain identifiers that are unique at different contexts.

Contexts could be global, host local or application local. All standard `uuid` tools are included in this one: `UUID`, `uuid1()`, `uuid3()`, `uuid4()`, `uuid5()`, `getnode()` and standard UUIDs constants *NAMESPACE_DNS*, *NAMESPACE_URL*, *NAMESPACE_OID* and *NAMESPACE_X500*.

This module also contains:

- *str_uuid()*: Return a string with a GUID representation, random if the argument is True, or a host ID if not.

New in version 1.7.0.

`xoutil.values.ids.`**`str_uuid`**`(`*random=False*`)`
Return a “Global Unique ID” as a string.

Parameters **`random`** – If True, a random uuid is generated (does not use host id).

`xoutil.values.simple` – Simple or internal coercers

Simple or internal coercers.

With coercers defined in this module, many of the `xoutil.string` utilities could be deprecated.

In Python 3, all arrays, not only those containing valid byte or unicode chars, are buffers.

`xoutil.values.simple.`**`ascii_coerce`**`(`*arg*`)`
Coerce to string containing only ASCII characters.

Convert all non-ascii to valid characters using unicode 'NFKC' normalization.

`xoutil.values.simple.`**`ascii_set_coerce`**`(`*arg*`)`
Coerce to string with only ASCII characters removing repetitions.

Convert all non-ascii to valid characters using unicode 'NFKC' normalization.

`xoutil.values.simple.`**`bytes_coerce`**`(`*arg*`)`
Encode an unicode string (or any object) returning a bytes buffer.

Uses the defined *encoding* system value.

In Python 2.x *bytes* coincide with *str* type, in Python 3 *str* uses unicode and *str* is different to *bytes*.

There are differences if you want to obtain a buffer in Python 2.x and Python 3; for example, the following code obtain different results:

```
>>> ba = bytes([65, 66, 67])
```

In Python 2.x is obtained the string `"[65, 66, 67]"` and in Python 3 `b"ABC"`. This function normalize these differences.

Name is used in named objects, see `name_coerce()` for more information.

See `str_coerce()` to coerce to standard string type, *bytes* in Python 2.x and unicode (*str*) in Python 3.

Always returns the *bytes* type.

New in version 1.7.0.

`xoutil.values.simple.`**`chars_coerce`**`(`*arg*`)`
Convert to unicode characters.

If *arg* is an integer between `0` and `0x10ffff` is converted assuming it as ordinal unicode code, else is converted with `unicode_coerce()`.

`xoutil.values.simple.`**`collection`**`(`*arg=nil*, *avoid=()*, *force=False*, *base=None*, *name=None*`)`
Coercer for logic collections.

Inner coercer returns the same argument if it is a strict iterable. In Python, strings are normally iterables, but never in our logic. So:

```
>>> collection('abc') is nil
True
```

This function could directly check an argument if it isn't `nil`, or returns a coercer using extra parameters:

Parameters

- **`avoid`** – a type or tuple of extra types to ignore as valid collections; for example:

```
>>> collection(avoid=dict)({}) is nil
True
>>> collection()({}) is nil
False
```

- **force** – if main argument is not a valid collection, it is are wrapped inner a list:

```
>>> collection(avoid=(dict,), force=True)({}) == [{}]
True
```

- **base** – if not `None`, must be the base to check instead of `Iterable`.
- **name** – decorate inner coercer with that function name.

`xoutil.values.simple.`**`decode_coerce`**(*arg*)
Decode objects implementing the buffer protocol.

`xoutil.values.simple.`**`encode_coerce`**(*arg*)
Encode string objects.

`xoutil.values.simple.`**`force_collection_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings and (<class '_abcoll.Mapping'>,) are not considered valid iterables in this case. A non iterable argument is wrapped in a list.

`xoutil.values.simple.`**`force_iterable_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings are not considered valid iterables in this case. A non iterable argument is wrapped in a list.

`xoutil.values.simple.`**`force_sequence_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings and (<class '_abcoll.Mapping'>,) are not considered valid iterables in this case. A non iterable argument is wrapped in a list.

`xoutil.values.simple.`**`isnot`**(*value*)
Create a coercer that returns *arg* if *arg* is not *value*.

`xoutil.values.simple.`**`iterable_coerce`**(*arg*)
Return the same argument if it is an iterable.

`xoutil.values.simple.`**`logic_collection_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings and (<class '_abcoll.Mapping'>,) are not considered valid iterables in this case.

`xoutil.values.simple.`**`logic_iterable_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings are not considered valid iterables in this case.

`xoutil.values.simple.`**`logic_sequence_coerce`**(*arg*)
Return the same argument if it is a strict iterable. Strings and (<class '_abcoll.Mapping'>,) are not considered valid iterables in this case.

`xoutil.values.simple.`**`lower_ascii_coerce`**(*arg*)
Coerce to string containing only lower-case ASCII characters.

Convert all non-ascii to valid characters using unicode 'NFKC' normalization.

`xoutil.values.simple.`**`lower_ascii_set_coerce`**(*arg*)
Coerce to string with only lower-case ASCII chars removing repetitions.

Convert all non-ascii to valid characters using unicode 'NFKC' normalization.

`xoutil.values.simple.`**`name_coerce`**(*arg*)
If *arg* is a named object, return its name, else *nil*.

Object names are always of *str* type, other types are considered invalid.

Generator objects has the special *__name__* attribute, but they are ignored and considered invalid.

`xoutil.values.simple.`**`not_false`**(*default*)

Create a coercer that returns *default* if *arg* is considered false.

See `not_false_coercer()` for more information on values considered false.

`xoutil.values.simple.`**`not_false_coercer`**(*arg*)

Validate that *arg* is not a false value.

Python convention for values considered True or False is not used here, our false values are only *None* or any false instance of `xoutil.symbols.boolean` (of course including *False* itself).

`xoutil.values.simple.`**`str_coerce`**(*arg*)

Coerce to standard string type.

bytes in Python 2.x and unicode (*str*) in Python 3.

New in version 1.7.0.

`xoutil.values.simple.`**`strict_string_coerce`**(*arg*)

Coerce to string only if argument is a valid string type.

class `xoutil.values.simple.`**`text`**

Return a nice text representation of one object.

text(obj='') -> text

text(bytes_or_buffer[, encoding[, errors]]) -> text

Create a new string object from the given object. If *encoding* or *errors* is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object text representation.

Parameters

- **`encoding`** – defaults to `sys.getdefaultencoding()`.
- **`errors`** – defaults to 'strict'.

Method join is improved, in order to receive any collection of objects, as variable number of arguments or as one iterable.

`chr_join`(*variable_number_args or iterable*) → text

Return a text which is the concatenation of the objects (converted to text) in argument items. The separator between elements is *S*.

Difference with `join()` is that integers between `0` and `0x10ffff` are converted to characters as unicode ordinal.

`join`(*variable_number_args or iterable*) → text

Return a text which is the concatenation of the objects (converted to text) in argument items. The separator between elements is *S*.

See `chr_join()` for other vertion of this functionality.

`xoutil.values.simple.`**`unicode_coerce`**(*arg*)

Decode a buffer or any object returning unicode text.

Uses the defined *encoding* system value.

In Python 2.x unicode has a special type different to *str* but in Python 3 coincide with *str* type.

Name is used in named objects, see `name_coerce()` for more information.

See `str_coerce()` to coerce to standard string type, *bytes* in Python 2.x and unicode (*str*) in Python 3.

New in version 1.7.0.

xoutil.web – Utils for Web applications

Utils for Web applications.

xoutil.web.**slugify**(*s*, *entities=True*, *decimal=True*, *hexadecimal=True*)

Convert a string to a slug representation.

Normalizes string, converts to lower-case, removes non-alpha characters, and converts spaces to hyphens.

Parts from http://www.djangosnippets.org/snippets/369/

```
>>> slugify("Manuel Vázquez Acosta")
'manuel-vazquez-acosta'
```

If *s* and *entities* is True (the default) all HTML entities are replaced by its equivalent character before normalization:

```
>>> slugify("Manuel V&aacute;zquez Acosta")
'manuel-vazquez-acosta'
```

If *entities* is False, then no HTML-entities substitution is made:

```
>>> value = "Manuel V&aacute;zquez Acosta"
>>> slugify(value, entities=False)
'manuel-v-aacute-zquez-acosta'
```

If *decimal* is True, then all entities of the form `&#nnnn` where *nnnn* is a decimal number deemed as a unicode codepoint, are replaced by the corresponding unicode character:

```
>>> slugify('Manuel V&#225;zquez Acosta')
'manuel-vazquez-acosta'

>>> value = 'Manuel V&#225;zquez Acosta'
>>> slugify(value, decimal=False)
'manuel-v-225-zquez-acosta'
```

If *hexadecimal* is True, then all entities of the form `&#nnnn` where *nnnn* is a hexdecimal number deemed as a unicode codepoint, are replaced by the corresponding unicode character:

```
>>> slugify('Manuel V&#x00e1;zquez Acosta')
'manuel-vazquez-acosta'

>>> slugify('Manuel V&#x00e1;zquez Acosta', hexadecimal=False)
'manuel-v-x00e1-zquez-acosta'
```

String Ambiguity in Python

In Python there are three semantic types when handling character strings:

1. Text: by nature can be part of internationalization processes. See Unicode for a standard for the representation, and handling of text in most of the world's writing systems.

In Python 2 there is an special type unicode to process **text**, but sometimes `str` is also used encoding the content; but in Python 3 `str` is always represented as **Unicode**.

2. Technical Strings: those used for for some special object names (classes, functions, modules, ...); the `__all__` definition in modules, identifiers, etc. Those values most times requires necessarily to be instances of `str` type. Try next in Python 2:

```
>>> class Foobar(object):
...     pass
>>> Foobar.__name__ = u'foobar'
TypeError: can only assign string to xxx.__name__, not 'unicode'
```

   In Python 2 `str` and `bytes` are synonymous; but in Python 3 are different types and `bytes` is exclusively used for **binary strings**.

3. Binary Strings: binary data (normally not readable by humans) represented as a character string. In Python 3 the main built-in type for this concept is `bytes`.

Mismatch Semantics Comparison

In Python 2 series, equal comparison for *unicode* an *str* types don't ever match. The following example fails in that version:

```
>>> s = 'λ'
>>> u = u'λ'
>>> u == s
False
```

Also a `UnicodeWarning` is issued with message "Unicode equal comparison failed to convert both arguments to Unicode - interpreting them as being unequal.

To correctly compare, use the same type. For example:

```
>>> from xoutil.eight.text import force
>>> force(s) == force(u)
True
```

Compatibility Modules

Xoutil has a Python 2 and 3 compatibility package named *`eight`*. So these issues related to ambiguity when handling strings (see Text versus binary data) are dealt in the sub-modules:

- *`text`*: tools related with *text handling*. In Python 2 values are processed with unicode and in Python 3 with standard `str` type.
- *`string`*: tools that in both versions of Python always use standard `str` type to fulfills *technical strings semantics*.

These modules can be used transparently in both Python versions.

Encoding Hell

To represent a entire range of characters is used some kind of encoding system. Maybe the trending top is the UTF (Unicode Transformation Format) family.

This complex diversity, even when strictly necessary for most applications, represents an actual "hell" for programmers.

For more references see `codecs` standard module. Also the `xoutil.future.codecs`, and `xoutil.eight.text` extension modules.

Changelog

1.8 series

Unreleased. Release 1.8.0

- Remove deprecated `xoutil.objects.metaclass`, use `xoutil.eight.meta.metaclass()` instead.
- Several modules are migrated to `xoutil.future`:
  - `types`.
  - `collections`.
  - `datetime`.
  - `functools`.
  - `inspect`.
  - `codecs`.
  - `json`.
  - `threading`.
  - `subprocess`.
  - `pprint`.
  - `textwrap`.
- Add function `xoutil.deprecation.import_deprecated()`, `inject_deprecated()` can be deprecated now.
- Add function `xoutil.deprecation.deprecate_linked()` to deprecate full modules imported from a linked version. The main example are all sub-modules of `xoutil.future`.
- Add function `xoutil.deprecation.deprecate_module()` to deprecate full modules when imported.
- Remove the module `xoutil.string` in favor of:
  - `xoutil.future.codecs`: Moved here functions `force_encoding()`, `safe_decode()`, and `safe_encode()`.
  - `xoutil.eight.string`: Technical string handling. In this module:
    * `force()`: Replaces old `safe_str`, and `force_str` versions.
    * `safe_join()`: Replaces old version in `future` module. This function is useless, it's equivalent to:

      ```
      force(vale).join(force(item) for item in iterator)
      ```

    * `force_ascii()`: Replaces old `normalize_ascii`. This function is safe and the result will be of standard `str` type containing only equivalent ASCII characters from the argument.

  - *xoutil.eight.text*: Text handling, strings can be part of internationalization processes. In this module:
    * *force()*: Replaces old `safe_str`, and `force_str` versions, but always returning the text type.
    * *safe_join()*: Replaces old version in `future` module, but in this case always return the text type. This function is useless, it's equivalent to:

      ```
      force(vale).join(force(item) for item in iterator)
      ```

  - `capitalize_word` function was completely removed, use instead standard method `word.capitalize()`.
  - Functions `capitalize`, `normalize_name`, `normalize_title`, `normalize_str`, `parse_boolean`, `parse_url_int` were completely removed.
  - `normalize_unicode` was completely removed, it's now replaced by *xoutil.eight.text.force()*.
  - `hyphen_name` was moved to *xoutil.cli.tools*.
  - `strfnumber` was moved as an internal function of 'xoutil.future.datetime':mod: module.
  - Function `normalize_slug` is now deprecated. You should use now *slugify()*.
- Create __small__ protocol for small string representations, see `xoutil.string.small()` for more information.
- Remove `xoutil.connote` that was introduced provisionally in 1.7.1.
- Module *xoutil.params* was introduced provisionally in 1.7.1, but now has been fully recovered.
  - Add function *issue_9137()* – Helper to fix issue 9137 (self ambiguity).
  - Add function *check_count()* – Checker for positional arguments actual count against constrains.
  - Add function *check_default()* – Default value getter when passed as a last excess positional argument.
  - Add function *single()* – Return true only when a unique argument is given.
  - Add function *keywords_only()* – Decorator to make a function to accepts its keywords arguments as keywords-only.
  - Add function *pop_keyword_arg()* – Tool to get a value from keyword arguments using several possible names.
  - Add class *ParamManager* – Parameter manager in a "smart" way.
  - Add class *ParamScheme* – Parameter scheme definition for a manager.
  - Add class *ParamSchemeRow* – Parameter scheme complement.
  - Remove `xoutil.params.ParamConformer`.
- Module *xoutil.values* was recovered adding several new features (old name `xoutil.cl` was deprecated).
- Add **experimental** module *xoutil.fp* for Functional Programming stuffs.
- Add **experimental** module `xoutil.tasking`.
- Remove deprecated module `xoutil.data`. Add `xoutil.objects.adapt_exception()`.
- Remove deprecated `xoutil.dim.meta.Signature.isunit()`.

1.7 series

2017-10-17 - 1.7.12

- `xoutil.datetime.EmptyTimeSpan` is now pickable.

2017-10-05. 1.7.11

Nothing yet

2017-09-21. 1.7.10

- Fix bug #6: `TimeSpan.overlaps` was incorrectly defined.
- Fix bug #5: `TimeSpan` can't have a *union* method.

2017-09-20. 1.7.9

- Deprecate `xoutil.dim.meta.Signature.isunit()`.
- Rename `xoutil.dim.meta.QuantityType` to `xoutil.dim.meta.Dimension`.
- Fix bug in `xoutil.datetime.TimeSpan`. `start_date` and `end_date` now return an instance of Python's `datetime.date` instead of a sub-class.

2017-09-19. 1.7.8

- Added module `xoutil.dim` – Facilities to work with concrete numbers.

2017-09-07. 1.7.7

- Fixed bug in `xoutil.datetime.date` that prevented to use `strftime()` in subclasses.
- Fixed bug in `xoutil.datetime.TimeSpan.valid()`.

2017-09-05. Release 1.7.6

- Fix a bug in `xoutil.datetime.TimeSpan` for Python 2. Representing a time span might fail with a 'Maximum Recursion Detected' error.

2017-09-05. Release 1.7.5

- Added `xoutil.datetime.TimeSpan`.
- Added the module `xoutil.infinity`.
- Added the keyword argument *on_error* to `xoutil.bound.until_errors()`.

2017-04-06. Release 1.7.4

- Added the argument *key* to `xoutil.iterators.delete_duplicates()`.
- Added the function `xoutil.iterators.iter_delete_duplicates()`.

2017-02-23. Release 1.7.3

- Add `xoutil.iterators.ungroup()`.
- Add `xoutil.future.datetime.get_next_month()`.

2017-02-07. Release 1.7.2

- Add `xoutil.bound.until()` and `xoutil.bound.until_errors()`.
- Fix issue that made `xoutil.uuid` unusable. Introduced in version 1.7.1, commit 58eb359.
- Remove support for Python 3.1 and Python 3.2.

2015-12-17. Release 1.7.1

- Add `xoutil.collections.PascalSet` and `xoutil.collections.BitPascalSet`.
- Add `xoutil.functools.lwraps()`.
- Add `xoutil.objects.multi_getter()`, `xoutil.objects.get_branch_subclasses()`, `xoutil.objects.fix_method_documentation()`.
- Add `xoutil.string.safe_str()`
- Remove long deprecated modules: `xoutil.aop` and `xoutil.proxy`.
- Deprecate *xoutil.html* entirely.
- The following modules are included on a *provisional basis*. Backwards incompatible changes (up to and including removal of the module) may occur if deemed necessary by the core developers:
  - `xoutil.connote`.
  - `xoutil.params`.

Fixes in 1.7.1.post1:

- Fix issue with both `xoutil.string.safe_decode()` and `xoutil.string.safe_encode()`.

  Previously, the parameter encoding could contain an invalid encoding name and the function could fail.

Fixes in 1.7.1.post2:

- Fix `xoutil.string.cut_suffix()`. The following invariant was being violated:

  ```
  >>> cut_suffix(v, '') == v  # for any value of 'v'
  ```

Warning: Due to lack of time, we have decided to release this version without proper releases of 1.7.0 and 1.6.11.

Unreleased. Release 1.7.0

This release was mainly focused in providing a new starting point for several other changes. This release is being synchronized with the last release of the 1.6.11 to allow deprecation messages to be included properly.

The following is the list of changes:

- The *defaults* `xoutil.objects.smart_copy()` has being made keyword only.
- Deprecates the `pop()` semantics, they shadow the `dict.pop()`. A new `pop_level()` is provided to explicitly pop a stack level. The same is done for the `pop()` method.
- Deprecates `xoutil.iterators.fake_dict_iteritems()`.
- Deprecates `xoutil.objects.metaclass` in favor for `xoutil.eight.meta.metaclass()`.

1.6 series

Unreleased. Release 1.6.11

This is the last release of the 1.6 series. It's being synchronized with release 1.7.0 to deprecate here what's being changed there.

- The *defaults* argument of `xoutil.objects.smart_copy()` is marked to be keyword-only in version 1.7.0.
- Fixes a bug in `xoutil.objects.smart_copy()`. If *defaults* was None is was not being treated the same as being False, as documented. This bug was also fixed in version 1.7.0.
- `xoutil.objects.metaclass()` will be moved to *xoutil.eight.meta* in version 1.7.0 and deprecated, it will be removed from `xoutil.object` in version 1.7.1.
- This release will be the last to support Python 3.1, 3.2 and 3.3. Support will be kept for Python 2.7 and Python 3.4.

2015-04-15. Release 1.6.10

- Fix `repr()` and `str()` issues with `xoutil.cli.Command` instances.

2015-04-03. Release 1.6.9

- The *defaults* argument in `xoutil.objects.smart_copy()` is now keyword-only.
- `xoutil.context` is now greenlet-safe without depending of *gevent*.

2015-01-26. Release 1.6.8

- Added `xoutil.records.date_reader()`.
- Added a forward-compatible `xoutil.inspect.getfullargspec()`.
- Now `contexts` will support gevent-locals if available. See the note in *the module documentation*.
- Minor fixes.

2014-12-17. Release 1.6.7

- Added the *strict* argument to `xoutil.records.datetime_reader()`.
- You may now install `xoutil[extra]` so that not required but useful packages are installed when xoutil is installed.

  For now this only includes `python-dateutil` that allows the change in `datetime_reader()`.

2014-11-26. Release 1.6.6

- Improved the `xoutil.string.normalize_slug()` by providing both valid and invalid chars.
- Added the `xoutil.string.normalize_ascii()`.

2014-10-13. Release 1.6.5

- Added the module `xoutil.records`.
- Deleted deprecated `xoutil.compat`.
- Deprecate the *xoutil.six*. It will removed in 1.7.0 (probably next release).

  Now xoutil requires *six* 1.8.0.

2014-09-13. Release 1.6.4

- Fix bug in `xoutil.fs.concatfiles()`: There were leaked opened files.

2014-08-05. Release 1.6.3

- Added the pre-release version of `xoutil.bound` module.

2014-08-04. Release 1.6.2

- Fix encoding issues in `xoutil.string.cut_prefix()` and `xoutil.string.cut_suffix()`.

  Previously this code failed:

  ```
  >>> from xoutil.string import cut_prefix
  >>> cut_prefix(u'-\xe1', '-')
  Traceback ...
    ...
  UnicodeEncodeError: 'ascii' ...
  ```

  Now both functions force its second argument to be of the same type of the first. See `xoutil.string.safe_decode()` and `xoutil.string.safe_encode()`.

2014-07-18. Release 1.6.1

- Added the *yield* parameter in `xoutil.fs.ensure_filename()`.
- Added the *base* parameter in `xoutil.modules.moduleproperty()`.
- Added the function `xoutil.fs.concatfiles()`.

2014-06-02. Release 1.6.0

- Changes the signature of *xoutil.names.nameof()*, also the semantics of the *full* parameter is improved.

  This is the major change in this release. Actually, this release has being prepared in sync with the release 1.5.6 (just a few days ago) to have this change passed while still keeping our versions scheme.

1.5 series

2014-05-29. Release 1.5.6

- Warn about a future backwards incompatible change in the behavior of *xoutil.names.nameof()*.

2014-05-13. Release 1.5.5

- UserList are now collections in the sense of `xoutil.types.is_collection()`.
- Python 3.4 added to the list of tested Python environments. Notice this does not makes any warrants about identical behavior of things that were previously backported from Python 3.3.

  For instance, the `xoutil.collections.ChainMap` has been already backported from Python 3.4, so it will have the same signature and behavior across all supported Python versions.

  But other new things in Python 3.4 are not yet backported to xoutil.
- Now *xoutil.objects.metaclass()* supports the `__prepare__` classmethod of metaclasses. This is fully supported in Python 3.0+ and partially mocked in Python 2.7.
- Backported `xoutil.types.MappingProxyType` from Python 3.3.
- Backported `xoutil.types.SimpleNamespace` from Python 3.4.
- Backported `xoutil.types.DynamicClassAttribute` from Python 3.4
- Added function *xoutil.iterators.delete_duplicates()*.
- Added parameter *ignore_underscore* to *xoutil.string.normalize_slug()*.
- Added module *xoutil.crypto* with a function for generating passwords.
- Fixed several bug in `xoutil.functools.compose()`.
- Makes *xoutil.fs.path.rtrim()* have a default value for the amount of step to traverse.

2014-04-08. Release 1.5.4

- Fix a bug in *xoutil.objects.extract_attrs()*. It was not raising exceptions when some attribute was not found and *default* was not provided.

  Also now the function supports paths like *xoutil.objects.get_traverser()*.
- *xoutil* contains now a copy of the excellent project six exported as `xoutil.six` (not documented here). Thus the compatibility module `xoutil.compat` is now deprecated and will removed in the future.

  There are some things that `xoutil.compat` has that `xoutil.six` does not. For instance, `six` does not include fine grained python version markers. So if your code depends not on Python 3 v Python 2 dichotomy but on features introduced in Python 3.2 you must use the `sys.version_info` directly.

  Notwithstanding that, `xoutil` will slowly backport several Python 3.3 standard library features to Python 2.7 so that they are consistently used in any Python up to 2.7 (but 3.0).

2014-04-01. Release 1.5.3

- Now *xoutil* supports Python 2.7, and 3.1+. Python 3.0 was not tested.
- Added a *strict* parameter to `xoutil.objects.smart_getter()`.
- New function `xoutil.objects.get_traverser()`.
- The function `xoutil.cli.app.main()` prefers its *default* parameter instead of the application's default command.

  Allow the `xoutil.cli.Command` to define a `command_cli_name` to change the name of the command. See `xoutil.cli.tools.command_name()`.

2014-03-03. Release 1.5.2

- Deprecated function `xoutil.objects.get_and_del_key()`. Use the `dict.pop()` directly.

  To have consistent naming, renamed `get_and_del_attr()` and `get_and_del_first_of()` to `popattr()` and `pop_first_of()`. Old names are left as deprecated aliases.
- Now `xoutil.functools.update_wrapper()`, `xoutil.functools.wraps()` and `xoutil.functools.lru_cache()` are Python 3.3 backports (or aliases).
- New module `xoutil.textwrap`.

2014-02-14. Release 1.5.1

- Added functions `xoutil.objects.dict_merge()`, `xoutil.types.are_instances()` and `xoutil.types.no_instances()`.
- Deprecated function `xoutil.objects.smart_getattr()`. Use `xoutil.objects.get_first_of()` instead.

2014-01-24. Release 1.5.0

- Lots of removals. Practically all deprecated since 1.4.0 (or before). Let's list a few but not all:
  - Both `xoutil.Unset` and `xoutil.Ignored` are no longer re-exported in `xoutil.types`.
  - Removes module `xoutil.decorator.compat`, since it only contained the deprecated decorator `xoutil.decorator.compat.metaclass()` in favor of `xoutil.objects.metaclass()`.
  - Removes `nameof` and `full_nameof` from `xoutil.objects` in favor of `xoutil.names.nameof()`.
  - Removes `pow_` alias of `xoutil.functools.power()`.
  - Removes the deprecated `xoutil.decorator.decorator` function. Use `xoutil.decorator.meta.decorator()` instead.
  - Now `get_module_path()` is documented and in module `xoutil.modules`.
- Also we have documented a few more functions, including `xoutil.fs.path.rtrim()`.
- All modules below `xoutil.aop` are in risk and are being deprecated.

1.4 series

- Adds `xoutil.datetime.daterange()`.
- Adds `xoutil.objects.traverse()`.
- Adds `xoutil.fs.makedirs()` and `xoutil.fs.ensure_filename()`.
- The *fill* argument in function `xoutil.iterators.slides()` now defaults to None. This is consistent with the intended usage of `Unset` and with the semantics of both `xoutil.iterators.continuously_slides()` and `xoutil.iterators.first_n()`.

  Unset, as a default value for parameters, is meant to signify the absence of an argument and thus only would be valid if an absent argument had some kind of effect *different* from passing the argument.
- Changes `xoutil.modules.customize()` API to separate options from custom attributes.
- Includes a *random* parameter to `xoutil.uuid.uuid()`.
- Deprecations and introductions:
  - Importing `xoutil.Unset` and `xoutil.Ignored` from `xoutil.types` now issues a warning.
  - New style for declaring portable metaclasses in `xoutil.objects.metaclass()`, so `xoutil.decorator.compat.metaclass()` is now deprecated.
  - Adds the module `xoutil.pprint` and function `xoutil.pprint.ppformat()`.
  - Adds the first version of package `xoutil.cli`.
  - Adds the *filter* parameter to functions `xoutil.objects.xdir()` and `xoutil.objects.fdir()` and deprecates *attr_filter* and *value_filter*.
  - Adds functions `xoutil.objects.attrclass()`, `xoutil.objects.fulldir()`.
  - Adds function `xoutil.iterators.continuously_slides()`.
  - Adds package `xoutil.threading`.
  - Adds package `xoutil.html` and begins the port of `xoutil.html.parser` from Python 3.3 to xoutil, so that a common implementation for both Python 2.7 and Python 3.3 is available.
- Bug fixes:
  - Fixes some errors with `classical` AOP weaving of functions in modules that where `customized`.
  - Fixes bugs with `xoutil.modules`: makes `xoutil.modules.modulemethod()` to customize the module, and improves performance.

2013-04-26. Release 1.4.0

- Refactors `xoutil.types` as explained in types-140-refactor.
- Changes involving `xoutil.collections`:
  - Moves SmartDict and SortedSmartDict from `xoutil.data` to `xoutil.collections`. They are still accessible from `xoutil.data`.
  - Also there is now a `xoutil.collections.SmartDictMixin` that implements the *update* behind all smart dicts in xoutil.
  - `xoutil.collections.StackedDict` in now a SmartDict and thus gains zero-level initialization data.
- Removals of deprecated, poorly tested, or incomplete features:

  - Removes deprecated `xoutil.decorators`. Use *`xoutil.decorator`*.
  - Removed `xoutil.iterators.first()`, and `xoutil.iterators.get_first()`.
  - Removed `xoutil.string.names()`, `xoutil.string.normalize_to_str()` and `xoutil.string.normalize_str_collection()`.
- Newly deprecated functions:
  - Deprecates `xoutil.iterators.obtain()`.
  - Deprecates `xoutil.iterators.smart_dict()` and *xoutil.data.smart_copy* in favor of *`xoutil.objects.smart_copy()`*.
- New features:
  - Introduces *`xoutil.iterators.first_non_null()`*.
  - Adds *`xoutil.objects.copy_class()`* and updates `xoutil.decorator.compat.metaclass()` to use it.
- Fixes a bug with *`xoutil.deprecation.deprecated()`* when used with classes: It changed the hierarchy and provoked infinite recursion in methods that use *super*.

1.3 series

- Removes deprecated module `xoutil.mdeco`.
- *`xoutil.context.Context`* now inherit from the newly created stacked dict class `xoutil.collections.StackedDict`. Whenever you enter a context a new level of the stacked dict is `pushed`, when you leave the context a level is <xoutil.collections.StackedDict.pop>‘:meth:.

  This also **removes** the *data* attribute execution context used to have, and, therefore, this is an incompatible change.
- Introduces `xoutil.collections.OpenDictMixin` and `xoutil.collections.StackedDict`.
- Fixes a bug in `xoutil.decorator.compat.metaclass()`: Slots were not properly handed.
- Fixes a bug with the simple `xoutil.collections.opendict` that allowed to shadow methods (even *__getitem__*) thus making the dict unusable.

1.2 series

2013-04-03. Release 1.2.3

- Bug fixes in `xoutil.proxy` and `xoutil.aop.classical`.

2013-03-25. Release 1.2.2

- Adds *`xoutil.bases`* - Implementations of base 32 and base 64 (numeric) representations.

2013-02-14. Release 1.2.1

- Loads of improvements for Python 3k compatibility: Several modules were fixed or adapted to work on both Python 2.7 and Python 3.2. They include (but we might have forgotten some):
  - *`xoutil.context`*.

  - xoutil.aop.basic.
  - *xoutil.deprecation*.
  - xoutil.proxy.

- Rescued *xoutil.annotate* and is going to be supported from now on.
- Introduced module xoutil.subprocess and function xoutil.subprocess.call_and_check_output().
- Introduced module xoutil.decorator.compat that enables constructions that are interoperable in Python 2 and Python 3.
- Introduced xoutil.iterators.zip(), xoutil.iterators.izip(), xoutil.iterators.map(), and xoutil.iterators.imap().

2013-01-04. Release 1.2.0

This is the first of the 1.2.0 series. It's been given a bump in the minor version number because we've removed some deprecated functions and/or modules.

- Several enhancements to *xoutil.string* to make it work on Python 2.7 and Python 3.2.

  Deprecates xoutil.string.normalize_to_str() in favor of the newly created xoutil.string.force_str() which is Python 3 friendly.

- Backwards incompatible changes in *xoutil.objects* API. For instance, replaces *getattr* parameter with *getter* in *xoutil.objects.xdir()* and co.
- Extracts decorator-making facilities from xoutil.decorators into xoutil.mdeco.
- Fixes in xoutil.aop.extended. Added parameters in xoutil.aop.classical.weave().
- Introduces *xoutil.iterators.first_n()* and deprecates xoutil.iterators.first() and xoutil.iterators.get_first().
- Removes the *zope.interface* awareness from *xoutil.context* since it contained a very hard to catch bug. Furthermore, this was included to help the implementation of *xotl.ql*, and it's no longer used there.

  This breaks version control policy since it was not deprecated beforehand, but we feel it's needed to avoid spreading this bug.

- Removed long-standing deprecated modules xoutil.default_dict, xoutil.memoize and xoutil.opendict.
- Fixes bug in xoutil.datetime.strfdelta(). It used to show things like '1h 62min'.
- Introduces xoutil.compat.class_type that holds class types for Python 2 or Python 3.

1.1 series

2012-11-01. Release 1.1.4

- Introduces xoutil.compat.iteritems_(), xoutil.compat.iterkeys_() and xoutil.compat.itervalues_().
- *execution context* are now aware of *zope.interface* interfaces; so that you may ask for a context name implementing a given interface, instead of the name itself.
- Improves xoutil.formatter documentation.

- Several fixes to `xoutil.aop.classical`. It has sudden backwards incompatible changes.
- *before* and *after* methods may use the **args, **kwargs* idiom to get the passed arguments of the weaved method.
- Several minor fixes: Invalid warning about Unset not in xoutil.types

2012-08-22. Release 1.1.3

- Adds function `xoutil.fs.rmdirs()` that removes empty dirs.
- Adds functions `xoutil.string.safe_join()`, `xoutil.string.safe_encode()`, `xoutil.string.safe_decode()`, and `xoutil.string.safe_strip()`; and the class `xoutil.string.SafeFormatter`.
- Adds function `xoutil.cpystack.iter_frames()`.

2012-07-11. Release 1.1.2

- Fixes all copyrights notices and chooses the PSF License for Python 3.2.3 as the license model for xoutil releases.
- All releases from now on will be publicly available at github.

2012-07-06. Release 1.1.1

- Improves deprecation warnings by pointing to the real calling filename
- Removes all internal use of simple_memoize since it's deprecated. We now use `lru_cache()`.

2012-07-03. Release 1.1.0

- Created the whole documentation Sphinx directory.
- Removed xoutil.future since it was not properly tested.
- Removed xoutil.annotate, since it's not portable across Python's VMs.
- Introduced module `xoutil.collections`
- Deprecated modules `xoutil.default_dict`, `xoutil.opendict` in favor of `xoutil.collections`.
- Backported `xoutil.functools.lru_cache()` from Python 3.2.
- Deprecated module `xoutil.memoize` in favor of `xoutil.functools.lru_cache()`.

1.0 series

2012-06-15. Release 1.0.30

- Introduces a new module :py‘xoutil.proxy‘:mod:.
- Starts working on the sphinx documentation so that we move to 1.1 release we a decent documentation.

2012-06-01. Release 1.0.29.

- Introduces *xoutil.iterators.slides* and *xoutil.aop.basic.contextualized*

2012-05-28. Release 1.0.28.

- Fixes normalize path and other details
- Makes validate_attrs to work with mappings as well as objects
- Improves complementors to use classes as a special case of sources
- Simplifies importing of legacy modules
- PEP8

2012-05-22. Release 1.0.27.

- Removes bugs that were not checked (tested) in the previous release.

2012-05-21. Release 1.0.26.

- Changes in AOP classic. Now you have to rename after, before and around methods to _after, _before and _around.

  It is expected that the signature of those methods change in the future.
- Introducing a default argument for `xoutil.objects.get_first_of()`.
- Other minor additions in the code. Refactoring and the like.

2012-04-30. Release 1.0.25.

- Extends the classical AOP approach to modules. Implements an extended version with hooks.
- 1.0.25.1: Makes classical/extended AOP more reliable to TypeError's in getattr. xoonko, may raise TypeError's for TranslatableFields.

2012-04-27. Release 1.0.24.

- Introduces a classical AOP implementation: xoutil.aop.classical.

2012-04-10. Release 1.0.23.

- Introduces decorators: xoutil.decorators.instantiate and xoutil.aop.complementor

2012-04-05. Release 1.0.22

- Allows annotation's expressions to use defined local variables. Before this release the following code raised an error:

```
>>> from xoutil.annotate import annotate
>>> x1 = 1
>>> @annotation('(a: x1)')
... def dummy():
...     pass
Traceback (most recent call last):
   ...
NameError: global name 'x1' is not defined
```

- Fixes decorators to allow args-less decorators

2012-04-03. Release 1.0.21

- Includes a new module `xoutil.annotate` that provides a way to place Python annotations in forward-compatible way.

How to contribute to xoutil

Testing

Running tests

xoutil uses *pytest* and *tox* for tests. We have a bundled version of *pytest* in the `runtests.py` scripts so for running tests in your environment you don't really have to install *pytest* and/or *tox*.

Given you have installed *xoutil* as development user-local package with:

```
$ python setup.py develop --user
```

You may run the tests with:

```
$ python runtests.py
```

Use the `-h` to show the *pytest* command line options.

If you have *tox* installed, then should have also Python 2.7, Python 3.5 and PyPy interpreters[1] installed and in your path to run the tests with *tox*. Having done so, you may run the tests with:

```
$ tox
```

This will run the tests suite in those three environments.

Writing tests

Testing was not introduced in *xoutil* until late in the project life. So there are many modules that lack a proper test suite.

To ease the task of writing tests, we chose *pytest*.

We use both normal tests ("à la pytest") and doctest. The purpose of doctests is testing the documentation instead of testing the code, which is the purpose of the former.

Most of our normal tests are currently simple functions with the "test" prefix and are located in the `tests/` directory.

Many functions that lacks are, though, tested by our use in other projects. However, it won't hurt if we write them.

Documentation

Since *xoutil* is collection of very disparate stuff, the documentation is hardly narrative but is contained in the docstrings of every "exported" element, except perhaps for module-level documentation in some cases. In these later cases, a more narrative text is placed in the `.rst` file that documents the module.

[1] See definitive list of needed Python interpreters in `tox.ini` file.

Versioning and deprecation

xoutil uses three version components.

The first number refers to language compatibility: *xoutil* 1.x series are devoted to keeping compatible versions of the code for both Python 2.7 and Python 3.2+. The jump to 2.x version series will made when *xoutil* won't support Python 2.7 any longer.

The second number is library major version indicator. This indicates, that some deprecated stuff are finally removed and/or new functionality is provided.

The third number is minor release number. Devoted to indicate mostly fixes to existing functionality. Though many times, some functions are merged and the old ones get a deprecation warning.

Occasionally, a fourth component is added to a release. This usually means a packaging problem, or bug in the documentation.

Module layout and rules

Many modules in *xoutil* contains definitions used in *xoutil* itself. Though we try to logically place every feature into a rightful, logical module; sometimes this is not possible because it would lead to import dependency cycles.

We are establishing several rules to keep our module layout and dependency quite stable while, at the same time, allowing developers to use almost every feature in xoutil.

We divide xoutil modules into 4 tiers:

1. Tier 0

   This tier groups the modules that **must not** depend from other modules besides the standard library. These modules implement some features that are exported through other xoutil modules. These module are never documented, but their re-exported features are documented elsewhere.

   Also the exported module `xoutil.eight` is this tier.

2. Tier 1

   In this tier we have:

   - `xoutil.decorator.meta`. This is to allow the definition of decorators in other modules.
   - `xoutil.names`. This is to allow the use of `xoutil.names.namelist` for the `__all__` attribute of other modules.
   - `xoutil.deprecation`. It **must not** depend on any other module besides `xoutil.eight`. Many modules in *xoutil* will use this module at import time to declare deprecated features.

3. Tier 2

   Modules in this tier should depend only on features defined in tiers 0 and 1 modules, and that export features that could be imported at the module level.

   This tier only has the `xoutil.modules`. Both `xoutil.modules.modulepropery()` and `xoutil.modules.modulemethod()` are meant be used at module level definitions, so they are likely to be imported at module level.

4. Tier 3

   The rest of the modules.

   In this tier, `xoutil.objects` and `xoutil.types` are kings. But in order to allow the import of other modules the following pair of rules are placed:

- At the module level only import from upper tiers.
- Imports from tier 3 are allowed, but only inside the functions that use them.

This entails that you can't define a function that must be a module level import, like a decorator for other functions. For that reason, decorators are mostly placed in the `xoutil.decorator` module.

The tiers above are a "logical suggestion" of how xoutil modules are organized and indicated how they might evolve.

List of contributors

If you're a contributor and you're not listed here, we appologize for that omission, and ask you to add yourself to the list.

- Medardo Rodríguez started this package and wrote most of it.
- Dunia Trujillo has fixed bugs, tested the software and also contributed code.
- Manuel Vázquez has contribute code and reorganize the package for the 1.1.x release series. He has contributed also to the documentation and docstring in reST format with doctests.

Copyright and Licence

Copyright (c) 2013-2017 Merchise Autrement [~º/~] and Contributors.

Copyright (c) 2012 Medardo Rodríguez.

This software is released under terms similar to the Python Software Foundation (PSF) licence for Python 3.2 as stated *below*.

Three modules inside this package are backports from Python 3.2.3's standard library and the PSF retains the copyright.

License Terms

This LICENSE AGREEMENT is between the Copyright Owner (Owner or Author), and the Individual or Organization ("Licensee") accessing and otherwise using xoutil 1.8.0 software in source or binary form and its associated documentation.

Subject to the terms and conditions of this License Agreement, the Owner hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use xoutil 1.8.0 alone or in any derivative version, provided, however, that Owner's License Agreement and Owner's notice of copyright, i.e., "Copyright (c) 2015 Merchise and Contributors" are retained in xoutil 1.8.0 alone or in any derivative version prepared by Licensee.

In the event Licensee prepares a derivative work that is based on or incorporates xoutil 1.8.0 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to xoutil 1.8.0.

The Owner is making xoutil 1.8.0 available to Licensee on an "AS IS" basis. THE OWNER MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, THE OWNER MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF xoutil 1.8.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

Index