advutils Documentation

Release 0.0.1.post4

David Toro

Mar 22, 2017

Contents

Contents:

CHAPTER 1

advutils package

Submodules

advutils.counters module

This module defines counters useful to generate numbers simulating digital counters

class advutils.counters.**BaseCounter**(*min_state*, *max_state*, *child=None*, *parent=None*, *invert_count=False*)

Bases: future.types.newobject.newobject

Base class to create counters

confirm()

Confirm whether in next count state is truncated

count

decrease()

Decreases counter Train state.

Returns None, use getting methods to know new states

decrease_confirm()

Decreases counter Train state and confirms when get_max_state() is depleted. That is when counters overall state change from 0 to get_max_state()-1.

Returns True if confirmation otherwise None, use getting methods to know new states

demote()

Takes child place.

Returns master counter, if not successful returns None

get_counter_train(*train=None*)

Get each instance of train.

Parameters **train** – list containing instances of parents

Returns list of instances from itself and children

get_max_state(*train=None*)
Get the overall maximum state of train.

Parameters **train** – list containing parent maximum states

Returns overall maximum state. Note: maximum state is never reached but this.get_max_state()-1

get_max_state_train(*train=None*)
Get each max_state of train.

Parameters **train** – list containing max_state of parents

Returns list of max_state from itself and children

get_state(*train=None*)
Get overall state of train.

Parameters **train** –

Returns overall state

get_state_train(*train=None*)
All states or state train.

Parameters **train** – list containing parent states

Returns list of state from itself and children

increase()
Increases counter Train state.

Returns None, use getting methods to know new states

increase_confirm()
Increases counter Train state and confirms when get_max_state() is depleted. That is when counters overall state change from get_max_state()-1 to 0.

Returns True if confirmation otherwise None, use getting methods to know new states

invert_count

invert_ranks()
Counters in Train are repositioned so that last counter takes first place and vice-versa.

Returns master counter of actual counter train

master

max_state

min_state

next()

promote()
Takes parent place.

Returns master counter, if not successful returns None

reset()
Resets itself and children.

Returns None

reset_children()
Resets only children.

Returns None

set_child(*child*)
Sets child safely by telling the other counter who is the new parent.

Parameters **child** – child counter

Returns None

set_max_state_train(*values*)
Sets safely a counter max_state train.

Parameters **value** – new train of max_state(list). If state exceeds max_state then state takes max_state-1.

Returns None

set_parent(*parent*)
Sets parent safely by telling the other counter who is the new child.

Parameters **parent** – parent counter

Returns None

set_state_train(*values*)
Sets safely a counter state train.

Parameters **values** – new train of states (list). If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0

Returns None

set_this_max_state(*value*)
Sets safely this counter max_state.

Parameters **value** – new max_state. If state exceeds max_state then state takes max_state-1

Returns None

set_this_state(*value*)
Sets safely this counter state.

Parameters **value** – new value state. If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0

Returns None

state

take_place(*new*)
Changes counter place with another counter.

Parameters **new** – counter which place is to be taken

Returns taken counter, if not successful returns None

yield_counter_train()
Get generator of each instance of train.

Returns generator of instances from itself and children

class advutils.counters.**Bit**(*state=1*, *name=None*, *description=None*, *child=None*, *parent=None*)
Bases: *advutils.counters.DigitCounter*

Simulates a bit counter from 0 to 1

max_state

class advutils.counters.**Bitstream**(*stream*, *invert_count=False*, *invert_order=False*, *order=None*, *default_class=<class 'advutils.counters.Bit'>*)

Bases: *advutils.counters.MechanicalCounter*

Simulates a bitstream of data

example:

```
a = Bitstream([1,1,1])
for i in a:
    print(i)
```

class advutils.counters.**DigitCounter**(*max_state*, *child=None*, *parent=None*, *invert_count=False*)

Bases: *advutils.counters.IntegerCounter*

Simulates a digit counter from 0 to 9

0

to

9

max_state

class advutils.counters.**IntegerCounter**(*max_state*, *child=None*, *parent=None*, *invert_count=False*)

Bases: *advutils.counters.BaseCounter*

Simulates an unsigned integer counter from 0 to Inf

0

to

Inf

max_state

set_state(*state*, *truncate=True*, *train=None*)

Sets safely this and all counters state train from the overall state.

Parameters

- **state** – overall state
- **truncate** – it is by default True. If True it calculates the states as if in a for loop:

```
for i in xrange(state):
    this.increase()
this.get_state()
```

If False it just stops to the last possible state before this.get_max_state()

- **train** – previously calculated train if this counter is not master ex: if this and master counters maxStates are T and M then extract the trains using get_max_state_train from both this and master, so that master elements reach until previous counter of this counter [M,10, ...,T-1]. Similarly, get_max_state method can be used as [master.get_max_state()/this.get_max_state()]

Returns

reached_overall_state

```
if truncate == True:
    assert reached_overall_state == state % a.get_max_state()
else:
    remaining = state - reached_overall_state
```

state

class advutils.counters.**MechanicalCounter**(*values*, *invert_count=False*, *invert_order=False*, *order=None*, *default_class=<class 'advutils.counters.DigitCounter'>*)

Bases: *advutils.counters.BaseCounter*

Simulates a mechanical counter. It is a wrapper over a train of counters. By default it uses a DigitCounter for each slot in the Train.

0	0	0	0	0

to

9	9	9	9	9

config(*values=None*, *invert_count=None*, *invert_order=False*, *order=None*)

Safely configure MechanicalCounter instance values

Parameters

- **values** – list of maximum states (excluded) or default_class objects inherited from counter
- **invert_count** – Default is False that begins to increase counters, else decrease counters
- **invert_order** – if True, take inverted values from order
- **order** – index order from Train of counters
- **default_class** – default class of any object to convert from values.

confirm()

Confirm whether in next count states are truncated

decrease()

Decreases counter Train state.

Returns None, use getting methods to know new states

decrease_confirm()

Decreases counter Train state and confirms when get_max_state() is depleted. That is when counters overall state change from self.min_state to get_max_state()-1.

Returns True if confirmation otherwise None, use getting methods to know new states

get_counter_train(*train=None*)

Get each instance of train virtually organized.

Parameters **train** – list containing instances of parents

Returns list of instances from itself and children

get_max_state(*train=None*)

Get the overall maximum state of train.

Parameters **train** – list containing parent maximum states

Returns overall maximum state. Note: maximum state is never reached but this.get_max_state()-1

get_max_state_train(*train=None*)
Get each max_state of train.

Parameters **train** – list containing max_state of parents

Returns list of max_state from itself and children

get_real_counter_train()
Get each instance of train physically organized.

Returns list of instances from itself and children

get_real_max_state_train()
Get each max_state of train.

Returns list of max_state from itself and children

get_real_state_train()
All states or state train.

Returns list of state from itself and children

get_state(*train=None*)
Get overall state of train.

Parameters **train** –

Returns overall state

get_state_train(*train=None*)
All states or state train.

Parameters **train** – list containing parent states

Returns list of state from itself and children

increase()
Increases counter Train state.

Returns None, use getting methods to know new states

increase_confirm()
Increases counter Train state and confirms when get_max_state() is depleted. That is when counters overall state change from get_max_state()-1 to self.min_state.

Returns True if confirmation otherwise None, use getting methods to know new states

invert_count

invert_link_order()
Counters hierarchy in Train are repositioned so that last counter takes first this.Order and vice-versa.

Returns master counter of actual counter train

..Note:

Same as self.invert_ranks()

invert_ranks()
Counters hierarchy in Train are repositioned so that last counter takes first this.Order and vice-versa.

Returns master counter of actual counter train

..Note:

```
Same as self.invert_link_order() but self.invert_ranks()
does not touch the links but the real positions
```

invert_real_order()
Counters in Train are repositioned so that last counter takes first place and vice-versa.

Returns master counter of actual counter trainl

link()
Linker method to link counters in the train so that increasing and decreasing methods affect all the counters in the due order.

max_state

next()

order

reset()
Resets itself and children.

Returns None

reset_children()
Resets itself and children.

Returns None

set_max_state_train(*values*)
Sets safely a counter max_state train.

Parameters **value** – new train of max_state(list). If state exceeds max_state then state takes max_state-1

Returns None

set_state(*state*, *truncate=True*, *train=None*)
Sets safely this and all counters state train from the overall state.

Parameters

- **state** – overall state
- **truncate** – it is by default True. If True it calculates the states as if in a for loop:

```
for i in xrange(state):
    this.increase()
return this.get_state()
```

  If False it just stops to the last possible state before this.get_max_state()

- **train** – previously calculated train if this counter is not master ex: if this and master counters maxStates are T and M then extract the trains using get_max_state_train from both this and master, so that master elements reach until previous counter of this counter [M,10, ...,T-1]. Similarly, get_max_state method can be used as [master.get_max_state()/this.get_max_state()]

Returns

reached_overall_state

```
if truncate == True:
    assert reached_overall_state == state % a.get_max_state()
else:
    remaining = state - reached_overall_state
```

set_state_train(*values*)
Sets safely a counter state train.

Parameters **values** – new train of states (list). If value exceeds max_state then value takes max_state-1 or if value is less than self.min_state it takes self.min_state

Returns None

set_this_max_state(*value*)
Sets safely this counter max_state.

Parameters **value** – new max_state. If state exceeds max_state then state takes max_state-1

Returns None

set_this_state(*value*)
Sets safely this counter state.

Parameters **value** – new value state. If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0

Returns None

state

train

yield_counter_train()
Yield each instance of train virtually organized.

Returns generator of instances from itself and children

class advutils.counters.**NumericalCounter**
Bases: future.types.newobject.newobject

Simulates a signed integer counter from -Inf to Inf

advutils.counters.**decima2base**(*decimal*, *base*, *array=None*)
Convert from decimal to any base.

Parameters

- **decimal** – number in base 10
- **base** – base to convert to e.g. 2 for binary, 16 for hexadecimal
- **array** – control list

Returns list of number in converted base

advutils.eventqueue module

This module define event classes for queueing tasks

class advutils.eventqueue.**Event**(*doc=None*)
Bases: future.types.newobject.newobject

Class defining an event

sender(*obj*, *objtype=None*)

class advutils.eventqueue.**EventHandler**(*event*, *sender*)
Bases: future.types.newobject.newobject

Class to handle an Event instance

append(*func*)
Add new event handler function.

Event handler function must be defined like func(sender, earg). You can add handler also by using '+=' operator.

fire(**args*, ***kwargs*)
Fire event and call all handler functions. You can call EventHandler object itself like *self(*args,**kwargs)* instead of *self.fire(*args,**kwargs)*.

remove(*func*)
Remove existing event handler function.

You can remove handler also by using '-=' operator.

advutils.password module

This module defines simple numerical and mapped passwords using letters in a deterministic way i.e. in order

class advutils.password.**Permutator**(*iterable*, *child=None*, *parent=None*, *invert_count=False*)
Bases: *advutils.counters.IntegerCounter*

Permute an iterable object

get_value()
Get current value from permutation state

get_value_train(*train=None*)
All states or state train.

Parameters **train** – list containing parent states

Returns list of state from itself and children

next()

set_value(*value*)
Set current value for permutation state

set_value_train(*values*)
Sets safely a counter state train.

Parameters **values** – new train of states (list). If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0.

Returns None

class advutils.password.**PermutatorTrain**(*values*, *invert_count=False*, *invert_order=False*, *order=None*, *default_class=<class 'advutils.password.Permutator'>*)
Bases: *advutils.counters.MechanicalCounter*

Permute a Train of iterable objects

get_real_value_train()
All states or state train.

Returns list of state from itself and children

get_value_train(*train=None*)
All states or state train.

Parameters **train** – list containing parent states

Returns list of state from itself and children

next()

set_real_value_train(*values*)
Sets safely a counter state train.

Parameters **values** – new train of states (list). If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0

Returns None

set_value_train(*values*)
Sets safely a counter state train.

Parameters **values** – new train of states (list). If value exceeds max_state then value takes max_state-1 or if value is less than 0 it takes 0.

Returns None

advutils.password.**chr2num**(*strings*)
converts a string to number

advutils.password.**deterministic**(*length=1*, *iterable='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX*

It generates permutations faster but it is not customizable. (it uses product from itertools). Deterministic function generates "iterable^length" combinations from iterable each row of "length" columns.

Parameters

- **length** – length of how many columns, or factor number
- **iterable** – list of items to permute

Returns itertools iterator

..Example:

factor = 1 assert len(list(deterministic(factor, iterable = COMBINATIONS))) == len(COMBINATIONS)**factor

advutils.password.**get_permutations**(*length=1*, *combination='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR*
invert_count=False, *invert_order=False*, *order=None*)
Like deterministic but customizable.

Parameters

- **length** – length of how many columns, or factor number
- **combination** – list of items to permute
- **invert_count** – Default is False, if True, take inverted index from combinations (see :param combinations)
- **invert_order** – Default is False, if True, take inverted index from order (see :param order)
- **order** – index order of columns

Returns PermutatorTrain object (it can be use as a generator)

advutils.password.**num2chr**(*nums*)
converts a number to string

advutils.password.**repeat_iterator**(*length=1*, *iterable='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV*
Repeats iterable "length" times.

Parameters

- **length** – length of how many columns or times to repeat
- **iterable** – list of items to repeat

Returns tuple of iterable rows and length columns

advutils.prettylogging module

ESC [0 m # reset all (colors and brightness) ESC [1 m # bright ESC [2 m # dim (looks same as normal brightness) ESC [22 m # normal brightness

FOREGROUND: ESC [30 m # black ESC [31 m # red ESC [32 m # green ESC [33 m # yellow ESC [34 m # blue ESC [35 m # magenta ESC [36 m # cyan ESC [37 m # white ESC [39 m # reset

BACKGROUND ESC [40 m # black ESC [41 m # red ESC [42 m # green ESC [43 m # yellow ESC [44 m # blue ESC [45 m # magenta ESC [46 m # cyan ESC [47 m # white ESC [49 m # reset

class advutils.prettylogging.**ANSIcolor**(*colors*)
Bases: future.types.newobject.newobject

Class defining ANSI color codes used in terminals

BLACK = '30'

BLACK_B = '40'

BLUE = '34'

BLUE_B = '44'

BRIGHT = 1

CYAN = '36'

CYAN_B = '46'

DIM = 2

GREEN = '32'

GREEN_B = '42'

MAGENTA = '35'

MAGENTA_B = '45'

NORMAL = 22

RED = '31'

RED_B = '41'

WHITE = '37'

WHITE_B = '47'

YELLOW = '33'

YELLOW_B = '43'

i = 47

paint(**args*, ***kwargs*)

class advutils.prettylogging.**CODE**(*name=None*, *level=0*, *colors=None*, *formatting='[{_code}]{_head}{_body}{_end}'*)

Bases: *advutils.BaseCopySupporter*

Class to define Logger codes like HIDDEN, DEBUG, ERROR, LOG, WARNING, IGNORE

codify(**args*, ***kwargs*)

Creates an instance of this CODE with default parameters

Parameters

- **text** – text to convert
- **newLine** – True to add new line at the end if needed
- **use_color** – True to use color
- **use_format** – True to use code formatting
- **kwargs** – additional kwargs to format

Returns formatted text

convert(*text*, *newLine=False*, *use_color=None*, *use_format=False*, ***kwargs*)

Convert text to CODE format and colors.

Parameters

- **text** – text to convert
- **newLine** – True to add new line at the end if needed
- **use_color** – True to use color
- **use_format** – True to use code formatting
- **kwargs** – additional kwargs to format

Returns formatted text

raw_msg()

class advutils.prettylogging.**CODElist**(*iterable*)

Bases: list

Especial list to hold CODE objects used in CodeMapper

append(*item*)

extend(*iterable*)

class advutils.prettylogging.**CodeLog**(*std_out=<open file '<stdout>', mode 'w'>*, *code_mapper=None*, *default_codes=None*, *use_colors=None*, *use_codes=None*)

Bases: future.types.newobject.newobject

Base Logger Class which supports CODE objects

accepted_code(*codes*)

return True if codes is accepted else False

convert(*text*, *codes=None*, *newLine=False*, ***kwargs*)

Convert text with code.

Parameters

- **text** – text to convert
- **codes** – codes to use for text
- **newLine** – True to add newline
- **kwargs** – additional arguments to pass to CODEs

Returns string of formatted text

convert_code(*codes=None*)

Filter accepted codes and adequate them to use.

Parameters **codes** – levels, codes or iterators with them.

Returns it gets None or list with only codes, no empty list (use if filtered)

default_codes

printline(*text*, *code=None*, ***kwargs*)

printlines(*lines*, *code=None*, ***kwargs*)

use_codes = None

use_colors = None

write(*text*, *code=None*, ***kwargs*)

class advutils.prettylogging.**CodeMapper**(*codes=None*, *refcodes=None*, *range=None*, *limit=True*)

Bases: future.types.newobject.newobject

Manage and convert CODE objects to other CODE objects

codes

get_by_index(*index*)

Parameters **index** –

Returns

get_by_level(*code*)

Parameters **code** –

Returns

get_by_reference(*code*)

Parameters **code** –

Returns

map_code(*code*)

Parameters **code** –

Returns

class advutils.prettylogging.**EmptyLogger**(*std_out=<open file '<stdout>', mode 'w'>*, *code_mapper=None*, *default_codes=None*, *use_colors=None*, *use_codes=None*)

Bases: *advutils.prettylogging.CodeLog*

Empty logger to not generate outputs

printline(*text*, *code=None*, ***kwargs*)

printlines(*lines*, *code=None*, ***kwargs*)

write(*text*, *code=None*, ***kwargs*)

class advutils.prettylogging.**Loggers**(*logs=None*, ***kwargs*)
Bases: future.types.newobject.newobject

Manage multiple loggers

post_setting(***kwargs*)
Assign keyword arguments to logs.

Parameters **kwargs** – keyword arguments

printline(*text*, *state=None*, ***kwargs*)

printlines(*lines*, *state=None*, ***kwargs*)

write(*text*, *state=None*, ***kwargs*)

class advutils.prettylogging.**SimpleLogger**(*std_out=<open file '<stdout>', mode 'w'>*, *code_mapper=None*, *default_codes=CODE(colors = <advutils.prettylogging.ANSIcolor object>*, *formatting = '[{_code}]{_head}{_body}{_end}'*, *name = 'LOG'*, *level = 0)*, *use_colors=None*, *use_codes=None*, *verbosity=None*)
Bases: *advutils.prettylogging.CodeLog*

Simple logger to print CODE objects

convert_code(*codes=None*)

Parameters **codes** – levels, codes or iterators with them.

Returns it gets None, list with only codes or empty list if filtered by verbosity

advutils.prettylogging.**formatter**(*format_string*, *kwargs*)
Default formatter used to format strings. Instead of *"{key}".format(**kwargs)* use *formatter("{key}", kwargs)* which ensures that no errors are generated when an user uses braces e.g. {}. Bear in mind that formatter consumes kwargs which in turns replaces an used key with empty string "". This can generate unusual behaviour if not well used.

advutils.prettylogging.**have_colours**(*stream*)
Detect if output console supports ANSI colors.

Parameters **stream** –

Returns

advutils.prettylogging.**scale**(*x*, *range*, *drange*)
From real coordinates get rendered coordinates.

Parameters

- **x** – source value
- **range** – (min,max) of x
- **drange** – (min,max) of sx

Returns scaled x (sx)

advutils.prettylogging.**scale_index**(*index*, *range*, *drange*, *circle=False*, *limit=False*)
Uses scale but adds support for indexing.

Parameters

- **index** –
- **range** –
- **drange** –
- **circle** –
- **limit** –

Returns

advutils.prettylogging.**separate**(*text*)
Process a text to get its parts.

Parameters **text** –

Returns [head,body,end]

advutils.randwords module

This module have some utilities to create random words from source

Created on Thu Jan 28 16:59:13 2016

@author: dev

advutils.randwords.**filewords**(*path=None*)
Get words from file.

Parameters **path** –

Returns

advutils.randwords.**generate**(*source=None*, *minwords=2*, *maxwords=5*, *rand=True*)
Generate random words from source

Parameters

- **source** –
- **minwords** –
- **maxwords** –
- **rand** –

Returns

advutils.randwords.**getwords**(*var*)
Get whole words from string that are formed with ascii letters.

Parameters **var** –

Returns

advutils.randwords.**sitewords**(*word_site=None*)
Get words from web.

Parameters **word_site** –

Returns

advutils.threader module

This module defines APIs for multitasking and queueing

class advutils.threader.**Designator**(*maxsize=0*)
Bases: Queue.PriorityQueue

Task Designator with priority queue

close()

Returns

get(*block=True*, *timeout=None*)
Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

isOpen()

Returns

class advutils.threader.**IterDecouple**(*iterable*, *processes=None*, *buffsize=0*, *handler=None*)
Bases: future.types.newobject.newobject

Decouple iterator from main thread and with processes.

close()

generator()
Generate detached data from self.iterable

join()
Wait until data is generated and consumed from self.iterable

next()

start()
Start generating data from self.iterable to be consumable from self.queue

class advutils.threader.**MultiProcessingAPI**(*spawn=False*)
Bases: future.types.newobject.newobject

Class to unify Multi processing and threading

Event()

Returns

Lock()

Returns

Pool(**args*, ***kwargs*)

Parameters

- **args** –
- **kwargs** –

Returns

Process(**args*, ***kwargs*)

Parameters

- **args** –
- **kwargs** –

Returns

Queue(**args*, ***kwargs*)

Parameters

- **args** –
- **kwargs** –

Returns

RLock()

Returns

Semaphore(**args*, ***kwargs*)

Parameters

- **args** –
- **kwargs** –

Returns

Thread(**args*, ***kwargs*)

Parameters

- **args** –
- **kwargs** –

Returns

decorate(*obj*)

Parameters **obj** –

Returns

manage(*obj*, **args*, ***kwargs*)

Parameters

- **obj** –
- **args** –
- **kwargs** –

Returns

class advutils.threader.**PriorityQueue**(*maxsize=0*)
Bases: Queue.PriorityQueue

Variant of Queue.PriorityQueue in that FIFO rule is kept inside the same priority number groups.

Entries are typically tuples of the form: (priority number, data).

class advutils.threader.**QueueCarrier**(*priority*)
Bases: *advutils.BaseCreation*

Base class Carrier used to convey data reliably in PriorityQueues

HIGHEST_PRIORITY = inf

LOWEST_PRIORITY = -inf

advutils.threader.**heappop**(*l*)
Consume last item from queue list (biggest carrier)

Parameters **l** – list queue

Returns last item from list

advutils.threader.**heappush**(*l*, *item*)
Append to queue with priority (where carriers are organized from smaller to biggest)

Parameters

- **l** – list queue
- **item** – Event

advutils.threader.**use_pool**(*func*, *iterable*, *workers=4*, *chunksize=1*)
Map function over iterable using workers.

Parameters

- **func** – function to use in processing
- **iterable** – iterable object
- **workers** – number of workers
- **chunksize** – number of chunks to process per thread

Returns

Module contents

class advutils.**BaseCopySupporter**
Bases: object

Base class for classes supporting cloning and spawning of itself. This same behaviour can be obtained using @copy_support decorator.

clone(**args*, ***kwargs*)
Clones instance with modifying parameters. Not that this creates a new instance.

Returns new_instance

get_arguments(*args=()*, *kwargs=None*, *onlykeys=False*, *onlyused=False*, *func=None*)
Get all function parameters configured in this instance mixed with additional arguments.

Parameters

- **self** – instance object
- **args** – arguments to modify
- **kwargs** – key arguments to modify
- **onlykeys** – return only key arguments
- **onlyused** – return only modified arguments
- **func** – function to get parameters from. If None it uses self.__init__

Returns args, kwargs

spawn(**args*, ***kwargs*)
Creates new Carrier of the same class with parameters of this instance.

Returns new_instance

class advutils.**BaseCreation**
Bases: object

Base class Carrier used to convey data reliably in PriorityQueues

creation_order

creation_time

creation_time_str
Creation time formated string

creation_time_struct
Creation time structure

exception advutils.**ClassNotAllowed**
Bases: exceptions.Exception

Exception to denote that given class is not allowed

exception advutils.**CorruptPersistent**
Bases: exceptions.EOFError, exceptions.IOError

Used for persistent data read from disk like pickles to denote it has been corrupted

class advutils.**NameSpace**
Bases: object

Used to store variables

exception advutils.**NoParserFound**
Bases: exceptions.Exception

Raise when no parser is found to use in a shell i.e to interpret user input

exception advutils.**NotCallable**
Bases: exceptions.Exception

Defines objectGetter error: given object is not callable.

exception advutils.**NotConvertibleToInt**
Bases: exceptions.ValueError

Exception to denote that value cannot be represented as int

exception advutils.**NotCreatable**
Bases: exceptions.Exception

Defines objectGetter error: objectGetter cannot create new object.

exception advutils.**TimeOutException**
Bases: exceptions.Exception

Raise an exception when a process surpasses the timeout

exception advutils.**TransferExeption**
Bases: exceptions.Exception

Raise an exception when transfered data is corrupt

exception advutils.**VariableNotAvailable**
Bases: exceptions.Exception

Exception for variable that is not available

exception advutils.**VariableNotDeletable**
Bases: exceptions.Exception

Exception for property not deletable

exception advutils.**VariableNotGettable**
Bases: exceptions.Exception

Exception for property not gettable

exception advutils.**VariableNotSettable**
Bases: exceptions.Exception

Exception for property not settable

advutils.**clone**(*self*, **args*, ***kwargs*)
Clones instance with modifying parameters. Not that this creates a new instance.

Returns new_instance

advutils.**copy_support**(*_map=None*, *convert=None*, *overwrite=()*)
Class decorator that fills spawn and clone methods. Be careful when using this decorator as it can give unexpected results if __init__ arguments and instance variables do not correspond to each other. Be very wary of property methods since they change the behaviour of calling a variable. You must ensure that classes abide the rules or decorator adapts to class to ensure good behaviour. It implements methods like "__repr__" to represent instance creation, "get_arguments" to get __init__ parameters with overwriting arguments, "clone" to copy instance with overwriting arguments and "spawn", to create new instance with constructor __init__ and its overwriting arguments.

example:

```
# simple example
@copy_support
class Klass(object)
    def __init__(data):
        self.data = data

arguments = {"data":"Original"}
instanceA = Klass(**arguments)
for key, val in arguments.items():
    assert val == getattr(instanceA,key)

instanceB = instanceA.spawn("Spawned")
instanceC = instanceA.clone("Cloned")
print(repr(instanceA))
print(repr(instanceB))
print(repr(instanceC))

# example mapping parameters
_map = {"data":"_data"}
@copy_support(_map = _map)
class Klass2(object)
    def __init__(data):
        self._data = data

arguments = {"data":"Original"}
instanceA2 = Klass2(**arguments)
```

```
for key, val in arguments.items():
    assert val == getattr(instanceA2,_map[key])

instanceB2 = instanceA.spawn("Spawned")
instanceC2 = instanceA.clone("Cloned")
print(repr(instanceA2))
print(repr(instanceB2))
print(repr(instanceC2))
```

Parameters

- **_map** – dictionary to map __init__ arguments with the instance variable names e.g. if in __init__ an argument is "data" but is assigned as "_data" then use {"data":"_data"}.
- **convert** – dictionary with the copying functions to add to class
- **overwrite** – force list of parameters to overwrite

Returns class

Note:

There is not risk from inheritance and they can be overwritten applying the decorator again. Classes with no __init__ or no parameters in them do not have risk of bad behaviour.

---

advutils.**get_arguments**(*self*, *args=()*, *kwargs=None*, *onlykeys=False*, *onlyused=False*, *func=None*)

Get all function parameters configured in this instance mixed with additional arguments.

Parameters

- **self** – instance object
- **args** – arguments to modify
- **kwargs** – key arguments to modify
- **onlykeys** – return only key arguments
- **onlyused** – return only modified arguments
- **func** – function to get parameters from. If None it uses self.__init__

Returns args, kwargs

advutils.**get_parameters**(*func*, *args=()*, *kwargs=None*, *onlykeys=False*, *onlyused=False*, *default=None*)

Get all function parameters with default values.

Parameters

- **func** – function to get parameters from
- **args** – arguments to modify
- **kwargs** – key arguments to modify
- **onlykeys** – return only key arguments
- **onlyused** – return only modified arguments
- **default** – default value to assign to key arguments

Returns args, kwargs

advutils.**spawn**(*self*, **args*, ***kwargs*)
Creates new Carrier of the same class with parameters of this instance.

Returns new_instance

advutils.**try_func**(*func*)
Sandbox to run a function and return its values or any produced error.

Parameters **func** – testing function

Returns results or Exception

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

T

U

V

W

Y