libutilitaspy Documentation

Release 0.1dev

Ernesto Posse

December 19, 2011

CONTENTS

libutilitaspy is a general-purpose library of data-structures, categories, patterns, aspects, testing and general utilities.

Contents:

CHAPTER

ONE

DOWNLOADING AND INSTALLING

1.1 Prerequisites

- Python 2.6 or later (available from http://www.python.org/getit/)

1.2 Installing using pip

The simplest and recommended method of installation is using pip.

To download and install, type [1] on a console window:

```
pip install https://sites.google.com/site/libutilitaspy/downloads/libutilitaspy-|version|.tar.gz
```

Note: If you don't have pip, then follow the instructions from this link: http://www.pip-installer.org/en/latest/installing.html

---

1.3 Installing using only distutils

Type [1] [2] on a console window:

```
cd /path/to/your/downloads/folder
wget https://sites.google.com/site/libutilitaspy/downloads/libutilitaspy-0.1dev.tar.gz
tar zxf libutilitaspy-0.1dev.tar.gz
cd libutilitaspy-0.1dev
python setup.py install
```

Note: It is preferable to install using pip, as it makes it easier to uninstall packages.

---

[1] You may need to type `sudo` in front of the `pip install ...` command (or `python setup.py install`)

[2] The `wget` command is a command on Unix-like systems (Linux, MacOSX) which downloads the distribution archive, and may not be available on all platforms. Alternatively you can just download and extract the archive in a folder of your choice. An alternative to `wget` is the `curl` command.

1.4 Uninstalling using pip

Type [1] on a console window:

```
pip uninstall libutilitaspy
```

1.5 Uninstalling using only distutils

1. Find out the Python prefix directory. You can do that by typing on the command line:

   ```
   $ python -c "import sys; print sys.prefix"
   ```

   For a standard central Python distribution, this is typically `/usr` or `/usr/local` on Linux [3] or `C:\Python26` on Windows, but it could be something else, for example if you are using `virtualenv`.

2. Find out the location of the Python `site-packages` directory. On Linux this is `<prefix>/lib/python<version>/site-packages` [4] and `<prefix>\lib\site-packages` on Windows.

   So for example, common locations are `/usr/lib/python2.6/site-packages` and `/usr/local/lib/python2.6/dist-packages`.

3. Go to the `site-packages` folder and remove the package and egg info, e.g.:

   ```
   $ cd /usr/lib/python2.6/site-packages
   $ rm -rf libutilitaspy*
   ```

4. On Linux, go to the `<prefix>/share` directory and remove the package folder. For example:

   ```
   $ cd /usr/share
   $ rm -rf libutilitaspy*
   ```

[3] Some distributions have Python on both `/usr` and `/usr/local` so you might have to check both.
[4] On Debian-based distributions such as Ubuntu, `site-packages` is usually called `dist-packages`.

CHAPTER

TWO

PACKAGES

2.1 data_structures package

This package contains modules that implement some common data-structures.

data_structures.maps	Provides utilities for functions, dictionaries, and associative tables.
data_structures.graphs	Provides an implementation of directed, labelled graphs.
data_structures.stacks	This module implements basic stacks.
data_structures.tries	Implements a trie data-structure (see http://en.wikipedia.org/wiki/Trie)
data_structures.heaps	Implements heaps.
data_structures.priority_queues	Implements priority queues using heaps.
data_structures.partitions	Provides utilities for creating set partitions.

2.1.1 libutilitaspy.data_structures.maps

Provides utilities for functions, dictionaries, and associative tables.

libutilitaspy.data_structures.maps.**dictunion**(*dict1*, *dict2*)

Returns a dictionary containing all key, value pairs of the given dictionaries.

Parameters

- **dict1** (*dict*) – Some dictionary
- **dict2** (*dict*) – Some dictionary

Returns The union of dict1 and dict2

Return type dict

class libutilitaspy.data_structures.maps.**Map**(*m*, *imp=None*)

Map objects generalize functions, dictionaries and associative tables. A map object behaves as both.

Map objects can be built from either:

- a `Mapping` object, or
- a `Callable` object

For example:

```
m1 = Map({1: 'a', 2: 'b', 3: 'a'})
m2 = Map(lambda x: 'a' if x in (1,3) else 'b')
```

It can be accessed with both dictionary and function call notation, e.g.:

```
y1 = m1[3]
y2 = m1(3)
```

Maps are mutable, even if defined by a function, e.g.:

```
m1[2] = 'c'
m2[2] = 'd'     # even though m2 was defined with a function rather than a dictionary.
```

If built from a `Mapping` object, the underlying `Map` implementation can be either:

- a dictionary (default), or
- a `libutilitaspy.data_structures.tries.Trie`

If built from a `Callable` object, the underlying `Map` implementation can be either:

- a function (default), or
- a memoized function

For example:

```
m1_1 = Map({1: 'a', 2: 'b', 3: 'a'}, 'dict')
m1_2 = Map({1: 'a', 2: 'b', 3: 'a'}, 'trie')
m2_1 = Map(lambda x: 'a' if x in (1,3) else 'b', 'func')
m2_2 = Map(lambda x: 'a' if x in (1,3) else 'b', 'mem')
```

Many combinations are possible. For example, you can create a map with a dictionary representation from a trie.

Note: A trie representation is useful when the mapping keys are sequences of hashable objects.

A memoized function representation is useful for callables which are pure functions (always return the same output on the same input), are called frequently and perform a significant amount of computation. The trade-off is that memory is used to cache previously computed results.

Parameters

- **m** (`Mapping` or `Callable`) – the mapping or callable object to make into a map.
- **imp** (*str*: for *Mapping* objects it can be 'dict' (default) or 'trie'; for *Callable* objects it can be 'func' (default) or 'mem'.) – The underlying implementation used

get_preimage(*value*)

Returns the *pre-image* of *value*, this is, the set of keys or indices or source values of the map, whose image is *value*.

Return type set

Note This method is available if the underlying implementation is a dictionary, a trie or a memoized function, but it is not available if it is a (non-memoized) function.

reflexive_closure()

Returns the reflexive closure of the map, i.e. the map extended with the identity map.

2.1.2 libutilitaspy.data_structures.graphs

Provides an implementation of directed, labelled graphs.

class libutilitaspy.data_structures.graphs.**Node**(*obj=None*)
Nodes of a graph. A node may have an object associated with it, and accessible through its obj atribute.

A node also has a set of incoming edges and a set of outgoing edges.

class libutilitaspy.data_structures.graphs.**Edge**(*source=None*, *target=None*, *label=None*)
Directed edges between nodes in a graph. An edge has source and target nodes, and possibly a label, which can be any object.

class libutilitaspy.data_structures.graphs.**Graph**(*nodes*, *edges*, *source=None*, *target=None*)
This class supports two basic, equivalent, styles of defining graphs:

1. **a graph is given by** (N, E, src, trg) **where**
   - N is a collection of nodes
   - E is a collection of edges
   - $src : E \to N$ is a map associating each edge e in E to its source node $src(e)$ in N
   - $trg : E \to N$ is a map associating each edge e in E to its target node $trg(e)$ in N
2. **a graph is given by** (N, E) **where**
   - N is a collection of nodes
   - E **is a collection of edges, where an edge** e **is a triple** (s, t, l) **with**
     - s is the source node of e,
     - t is the target node of e,
     - l is some label (any object)

In the first style, the connections of an edge are stored in the graph, whereas in the second style, each edge stores a reference to its source and target.

This implementation allows both styles. For example, using style 1) we have:

```
n1 = Node(1)
n2 = Node(2)
n3 = Node(3)
N = [n1, n2, n3]
e1 = Edge(label='a')
e2 = Edge(label='b')
e3 = Edge(label='a')
E = [e1, e2, e3]
src = Map({e1: n1, e2: n1, e3: n3})
trg = Map({e1: n2, e2: n3, e3: n3})
g = Graph(N, E, src, trg)
```

On the other hand, using style 2) we have:

```
n1 = Node(1)
n2 = Node(2)
n3 = Node(3)
N = [n1, n2, n3]
e1 = Edge(n1, n2, label='a')
e2 = Edge(n1, n3, label='b')
e3 = Edge(n3, n3, label='a')
E = [e1, e2, e3]
g = Graph(N, E)
```

In both cases you can access (where g is a Graph, e is an Edge, and n is a Node):

```
g.nodes         # this is a set of edges
g.edges         # this is a set of edges
g.source(e)     # this is a Node
g.target(e)     # this is a Node
e.source        # this is a Node
e.target        # this is a Node
n.incoming      # this is a set of edges
n.outgoing      # this is a set of edges
```

Invariants:

For any Graph g:

for every e in g.edges: e.source == g.source(e) and e.target == g.target(e) and e in e.source.outgoing and e in e.target.incoming

add_node(*node*)
Adds a node to the graph.

Parameters **node** (*Node*) – A node

add_edge(*edge*)
Adds an edge to the graph. If the source and/or target nodes of the edge are not already in the graph, they are also added.

Parameters **edge** (*Edge*) – An edge.

class libutilitaspy.data_structures.graphs.**GraphHomomorphism**(*source*, *target*, *nodemap*, *edgemap*)
A graph homomorphism $h : G_1 \to G_2$ between two graphs $G_1 = (N_1, E_1, src_1, trg_1)$ and $G_2 = (N_2, E_2, src_2, trg_2)$ is a pair of maps $h = (h_N, h_E)$ where $h_N : N_1 \to N_2$ and $h_E : E_1 \to E_2$ such that for all edges $e \in E_1$:

- $src_2(h_E(e)) = h_N(src_1(e))$, and
- $trg_2(h_E(e)) = h_N(trg_1(e))$

Equivalently, a graph homomorphism $h : G_1 \to G_2$ between two graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ is a pair of maps $h = (h_N, h_E)$ where $h_N : N_1 \to N_2$ and $h_E : E_1 \to E_2$ such that for all edges $e \in E_1$:

- if $e = (s, t, l) \in E_1$ then $h_E(e) = (h_N(s), h_N(t), l) \in E_2$.

map(*x*)

Parameters **x** (Node or Edge) – A node or edge in the source graph

Returns x's image

Return type Node if type(x) is Node, Edge if type(x) is Edge

2.1.3 libutilitaspy.data_structures.stacks

This module implements basic stacks. See http://en.wikipedia.org/wiki/Stack_(data_structure)

exception libutilitaspy.data_structures.stacks.**EmptyStack**
This exception is used to indicate that a stack is empty when attempting to pop an item.

class libutilitaspy.data_structures.stacks.**Stack**(*elements*=[])
This class implements basic stacks, supporting non-popping iteration over its items.

push(*item*)
Pushes item to the top of the stack.

pop()
Pops and returns the top of the stack. :returns: The top item in the stack. :raises EmptyStack: if the stack is empty.

top()
Returns the top of the stack without poping it. :returns: The top item in the stack. :raises EmptyStack: if the stack is empty.

isempty()

Returns True if the stack is empty, False otherwise.

next()
Obtains the next item in the stack, without removing it. :raises: StopIteration if the iterator reaches the bottom of the stack.

2.1.4 libutilitaspy.data_structures.tries

Implements a trie data-structure (see http://en.wikipedia.org/wiki/Trie)

Typical use:

- For tries without data:

```
t = Trie(['ab', 'ac', 'abc'])
try:
    value = t.search('ac')
except KeyError, e:
    print 'not found'
```

- For tries with data:

```
t = Trie({'ab':'data1', 'ac':'data2', 'abc':'data3'})
data = t.search('ac')

t.assign('ac', 'data4')     # changes the data associated to 'ac'
t.assign('acd', 'data5')  # adds 'acd' to t and associates 'data5' to it.
```

- Trees with data can also be constructed with lists of tuples, or generator expressions:

```
t = Trie([('ab',data1), ('ac', data2), ('abc', data3)])
```

or

```
t = Trie((k, data) for k in ['ab','ac',abc'])
```

- There is a mapping type syntax for these methods:

```
t['ac']
```

is equivalent to

```
t.search('ac')
```

and

```
t['ac'] = 'data6'
```

is equivalent to

```
t.assign('ac', 'data6')
```

class libutilitaspy.data_structures.tries.**Trie**(*sequences={}*)

A trie is a tree data-structure that is used to implement associative arrays or maps from sequences to some data. Typically the keys are strings, but they can be any sequence of hashable objects.

The data is stored in the tree's nodes, and the branches are labelled with the objects in the key sequence. Thus, searching for an item with a key *k = (k_1,k_2, .., k_n)* is done by following the path *k* from the root, and at each node *i* choosing the branch labelled with 'k_i'.

Nodes in the tree are tuples of the form

(*id*, *arrows*, *data*)

where *id* is a unique identifier within the trie (a natural number,) *arrows* is a dictionary with keys being the objects labelling the arrows from the node, and values being the target nodes, i.e. a pair (*key*, *value*) in this dictionary is really a pair (*obj*, *target-node*), so there is an arrow from this node to the target node labelled *obj*; and *data* is any additional data associated with this node.

The constructor creates a new trie with the given dictionary indexed by sequences. The keys are expected to be the sequences of hashable objects and the values are the data to be stored at the end node for the corresponding sequence.

Parameters **sequences** ((*sequence*, 'a) dict, where *sequence* is either a tuple, a list or any other sequence object, and 'a is any type.) – A dictionary mapping sequences to values to put in the trie.

assign(*key*, *value=None*)

Adds the (key,value) association to the trie.

Parameters

- **seq** – The key to store
- **value** – The associated value (any object).

search(*key*)

Looks for the key in the trie. It returns the data associated with the key, or raises a KeyError exception if the key is not in the trie.

Parameters **key** (*hashable seq, a sequence (tuple, list, etc.) of hashable objects*) – A key to search.

Returns the value object associated to the key if it is in the trie.

Raises KeyError if the key is not in the trie.

reset()

Resets the trie's pointer to the root node.

Post self.pointer == self.root

step(*obj*)

Moves the trie's pointer to the node following the branch labelled with the given object.

Parameters **obj** (*hashable*) – an label in one of the trie's branches

Returns the data contained in the node after the step is taken.

2.1.5 libutilitaspy.data_structures.heaps

Implements heaps.

2.1.6 libutilitaspy.data_structures.priority_queues

Implements priority queues using heaps.

class libutilitaspy.data_structures.priority_queues.**PriorityQueue**(*data=None*, *ascending=False*)

A priority queue is a queue of items such that its elements are extracted in order of their priority (see http://en.wikipedia.org/wiki/Priority_queue).

The priority of the items determines how they are compared, thus this class assumes that if two objects a and b are to be put in a queue with ascending order, a < b if and only if a's priority is higher than b's. Dually, if they are put in a queue with descending order, a > b if and only if a's priority is higher than b's.

The constructor creates a new queue from a given sequence.

Parameters

- **data** (*MutableSequence*) – an initial sequence of comparable objects.
- **ascending** (*bool*) – True if the items are sorted in ascending order, False for descending order.

Todo map data items to PQE

2.1.7 libutilitaspy.data_structures.partitions

Provides utilities for creating set partitions.

@author: eposse

libutilitaspy.data_structures.partitions.**create_partition**(*l*)

Given an iterable object l of comparable/hashable objects, return a partition table, namely the quotient of the set of elements in the list with respect to the equivalence relation given by the equality operation between the objects (as implemented by the __eq__ method).

This partition is a dictionary p, whose keys are elements of the list, and for a given element x from l, the corresponding value p[x] is the list of all elements y in l such that x == y.

The partition is computed as follows:

> For every item x in the list l, check if there is already an entry p[x]. More precisely, check if there is already a key y in p such that x == y. If so, add x to the list p[y]. If not, create a new entry p[x] and set it to be the list [x].

The result is computing by taking advantage of the semantics of dictionaries in Python: the __getitem__ operation invoked when accessing the entry p[x], first computes the hash value of x to get a quick access to the underlying table, and then, if there is a key x' in p with that hash value, Python checks to see if id(x) is id(x'). If so, they are the same object. If not, Python tries to compare with equality: x == x'. If they are equivalent, they are assumed to be the same key, because all hashable objects x and x' which compare equal, x == x', must have hash(x) == hash(x').

Parameters: l: 'a list

Returns:

('a, 'a list) dict A dictionary mapping elements of l to their equivalence class

Preconditions:

All elements in l must be hashable and comparable: all(hashable(x) for x in l)

libutilitaspy.data_structures.partitions.**create_partition_eq**(*l*, *eq*)
Like create_partition, but the partition is created with respect to the equivalence relation eq, rather than the __eq__ method of objects in l.

Parameters: l: 'a list eq: 'a * 'a -> bool

Returns:

('a, 'a list) dict A dictionary mapping elements of l to their equivalence class

Preconditions: eq is an equivalence relation (more precisely, the characteristic function of an equivalence relation)

2.2 categories package

This package contains modules that implement some category-theoretical concepts (see http://en.wikipedia.org/wiki/Category_theory).

categories.categories	This module is provides a representation for categories from Category Theory.
categories.diagrams	Author: Ernesto Posse
categories.limits	Author: Ernesto Posse
categories.colimits	Author: Ernesto Posse
categories.finite_sets	Author: Ernesto Posse

2.2.1 libutilitaspy.categories.categories

This module is provides a representation for categories from Category Theory.

For a definition of categories, see http://en.wikipedia.org/wiki/Category_theory

and http://en.wikipedia.org/wiki/Category_(mathematics)

class libutilitaspy.categories.categories.**Object**(*obj=None*, *category=None*)
Instances of this class represent objects in some category.

class libutilitaspy.categories.categories.**Arrow**(*source*, *target*, *label=None*, *category=None*)
Instances of this class represent arrows or morphisms between objects in some category.

dual()
Warning: this does not compute inverse functions.

compose(*other*)
Returns the composition of this arrow with another arrow, where this arrow goes first.

Parameters **other** (*Arrow*) – some arrow

Returns self.category.composite(self, other)

class libutilitaspy.categories.categories.**Category**(*object_class*, *arrow_class*)
Instances of this class are intended to represent categories.

A category consists of:

1. a class O of objects $A, B, ...$
2. **a class M of arrows (or morphisms) between objects** $f, g, ...$ where $f : A \to B$ is an arrow with source A and target B, in which case we define $src(f) = A$ and $trg(f) = B$
3. a composition operation $\circ$ between arrows
4. an 'identity' arrow id_A for each object A

And it must satisfy the following:

1. for every pair of arrows f and g, if the target of f is the source of g, then their composition, written $g \circ f$, exists, and is an arrow whose source is the source of f and whose target is the target of g. In short:

   if $f : A \to B \in M$ and $g : B \to C \in M$ then $g \circ f : A \to C \in M$

2. for each object A, the identity arrow $id_A : A \to A$ is the identity with respect to composition. This is,

   (a) for any arrow $f : A \to B \in M$, $f \circ id_A = f$

   (b) for any arrow $g : B \to A \in M$, $id_A \circ g = g$

3. Composition is associative: for any arrows $f : A \to B$, $g : B \to C$, $h : C \to D$ $f \circ (g \circ h) = (f \circ g) \circ h$

Instances of the Category class are intended to implement these categories, where objects are any Python objects and arrows are instances of the Arrow class (or any subclass).

This class is intended to be subclassed. In particular, a subclass should implement the *composition* and *identity* methods. Then to obtain the composition, a user calls the `composite()` method and to obtaind the identity, the user calls the `ident()` method of this class.

The `composite()` method is normally called via the `Arrow.__mul__()` method so if $f : A \to B \in M$ and $g : B \to C \in M$ then we can obtain $g \circ f : A \to C \in M$ by calling:

```
g * f
```

instead of:

```
cat.composite(f,g)
```

The constructor creates a category whose objects and arrows belong to the given classes.

Parameters

- **object_class** (subclass of `Object`) –
- **arrow_class** (subclass of `Arrow`) –

composite(*f*, *g*, *memoize=True*)

Returns the arrow g * f. Note that this intends to be f first, then g: f: A -> B and g: B -> C, then g * f == f >> g : A -> C

Note: it does not apply the composition, just computes and returns the composite arrow.

ident(*obj*, *memoize=True*)

Returns the identity arrow of the object given. This method however, does not check that the arrow is indeed the identity.

hom(*A*, *B*)

Should return the set of all arrows between A and B

colimit(*diagram*)

Abstract method: should compute the colimit of the diagram in the category.

class libutilitaspy.categories.categories.**FiniteCategory**(*object_class*, *arrow_class*, *objects=None*, *arrows=None*)

Instances of this subclass of Category make the assumption that the sets of objects and arrows are finite.

hom(*A*, *B*)

Should return the set of all arrows between A and B

close()

Compute the closure: generates all compositions and identities.

TODO: check whether this indeed computes the closure: add_edge adds new edges to self.arrows, but does this guarantee that all newly added edges will be traversed? If not, we may replace it with something like this:

arrow_list = list(self.arrows) new_arrows = [] for f in arrow_list:

for g in f.target.outgoing: new_arrow = self.compose(f, g) new_arrows.append(new_arrow) arrow_list.append(new_arrow)

for f in new_arrows: self.add_edge(f)

2.2.2 libutilitaspy.categories.diagrams

Author: Ernesto Posse

Description: This module contains the categorical definitions of diagrams. For a formal definition see:

http://en.wikipedia.org/wiki/Diagram_(category_theory)

Note: it includes the algorithms that compute pullbacks in terms of products and equalizers, and pushouts in terms of coproducts and coequalizers. Hence, a specific category does not need to implement either a pullback nor a pushout method; it only needs to implement product, coproduct, equalizer and coequalizer. In order to do that, the client must implement subclasses or Pair with methods product and coproduct, and ParallelArrows with methods equalizer and coequalizer

2.2.3 libutilitaspy.categories.limits

Author: Ernesto Posse

Description: This module contains the categorical definitions of limits. For a formal definition see:

http://en.wikipedia.org/wiki/Limit_(category_theory)

Note: it contains the only the classes that represent limits, not the algorithms that compute them. For these, see the 'diagrams' module.

In particular, the argument to the Limit constructor is not the diagram of which the limit is a limit of. Instead the arguments are the elements that form a limit, namely an object and a family of arrows, together with the function that computes the unique arrow from any other cone:

Limit(object, arrows, unique_arrow_func, category)

To obtain the limit of a diagram, one must create an instance of diagram and invoke the limit method on that instance. This method must return a Limit instance.

New: for ease of use, I added a functional interface to compute limits.

2.2.4 libutilitaspy.categories.colimits

Author: Ernesto Posse

Description: This module contains the categorical definitions of colimits. For a formal definition see:

http://en.wikipedia.org/wiki/Limit_(category_theory)

Note: it contains the only the classes that represent colimits, not the algorithms that compute them. For these, see the 'diagrams' module.

In particular, the argument to the CoLimit constructor is not the diagram of which the colimit is a colimit of. Instead the arguments are the elements that form a colimit, namely an object and a family of arrows, together with the function that computes the unique arrow to any other cone:

CoLimit(object, arrows, unique_arrow_func, category)

To obtain the colimit of a diagram, one must create an instance of diagram and invoke the colimit method on that instance. This method must return a CoLimit instance.

New: for ease of use, I added a functional interface to compute colimits.

2.2.5 libutilitaspy.categories.finite_sets

Author: Ernesto Posse Description:

Category of finite sets

Typical use:

Cat = DefaultFiniteSetCategory S1 = Set(['a','b','c']) S2 = Set([0,1]) f = Function(S1, S2, {'a':0, 'b':1, 'c':0}) P = S1 * S2 # equivalent to P = DefaultFiniteSetCategory.cartesian_product(S1,S2) D = S1 + S2 # equivalent to P = DefaultFiniteSetCategory.disjoint_union(S1,S2) x = P.proj1(('b',0)) y = S.inj1('c') z = S.inj2(1) u = P.unique_arrow(D.vertex) w = u(('a',1)) g = Function(S1, S2, {'a':1, b:'1', 'c':0}) Q1 = Cat.equalizer(f, g) Q2 = Cat.coequalizer(f, g) S3 = Set(['x','y','z']) h = Function(S3, S2, {'x':0, 'y':0, 'z':0}) P2 = Cat.pullback(f, g) k = Function(S1, S3, {'a':'y', 'b':'y', 'c':'x'}) P3 = Cat.pushout(f, k)

Cat = FiniteSetCategory() Cat.add_object(S1) Cat.add_object(S2) Cat.add_arrow(f)

2.3 patterns package

This package contains modules that implement some base classes used in common design patterns.

`patterns.observer`	This module implements the base classes of the *observer pattern* [GoF94].

2.3.1 libutilitaspy.patterns.observer

This module implements the base classes of the *observer pattern* [GoF94].

class `libutilitaspy.patterns.observer.`**`Observer`**

An *observer* (also called *listener*) is any object which can be registered with some `Observable` object to be notified whenever the Observable object is updated.

`notify`(**args*, ***kwargs*)

Method executed whenever any Observable where self is registered is updated. It should be overriden by a subclass.

class libutilitaspy.patterns.observer.**Observable**

An *observable* object is any object that has a list of registered Observer objects which are notified whenever the observable object is updated (a relevant change of state occurs).

register(*observer*)

Registers the observer object.

Parameters **observer** (*Observer*) – The object to register.

Post after(self.observers) == before(self.observers) + [observer]

deregister(*observer*)

Unregisters the observer object.

Parameters **observer** (*Observer*) – The object to register.

Pre observer in self.observers

Post after(self.observers) == before(self.observers) - [observer]

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

Notifies all registered observers.

2.4 aspects package

This package contains modules that implement mechanisms for aspect-oriented programming.

aspects.core	This module implements an aspect weaver.
aspects.logger	This module implements a general Logger aspect for logging method calls.
aspects.memoizer	This module implements a general Memoizer aspect for memoizing (caching) method results.

2.4.1 libutilitaspy.aspects.core

This module implements an aspect weaver.

It provides mainly two things:

- A superclass Aspect, to be the parent class of aspect classes. Each aspect contains a *pointcut* describing the classes and methods to which the aspect will be applied, and *advice* in the form of *before* and *after* methods, to be executed in any join-point in the pointcut.
- An *aspect weaver* in the form of meta-class factory, which generates a meta-class of any class to be affected (weaved) by a given list of aspects.

Typical use:

Define aspect classes in some modules. For example:

Module *myaspect1.py*:

```
from libutilitaspy.aspects import Aspect

class MyAspect1(Aspect):
    classes = 'SomeClass[0-9]+'     # a regular expression defining the (names of the) classes to whi
    methods = 'some_method'         # a regular expression defining the (names of the) methods to whi
    def before(self, klass, method, obj, *args, **kwargs):
        print "(aspect1) before"
        # do something...
    def after(self, klass, method, obj, retval, exc_type, exc_val, traceback):
```

```
        print "(aspect1) after"
        # do something...
```

Module *myaspect2.py*:

```
from libutilitaspy.aspects import Aspect

class MyAspect2(Aspect):
    classes = 'SomeClass[0-9]+'     # a regular expression
    methods = '.*_method'           # a regular expression
    def before(self, klass, method, obj, *args, **kwargs):
        print "(aspect2) before"
        # do something...
    def after(self, klass, method, obj, retval, exc_type, exc_val, traceback):
        print "(aspect2) after"
        # do something...
```

Now, you create the aspect weaver which is a metaclass, as follows:

Module *metaclassconfig.py*:

```
from libutilitaspy.aspects.core import WeaverMetaClassFactory
from myaspect1 import MyAspect1
from myaspect2 import MyAspect2

MyMetaClass = WeaverMetaClassFactory(MyAspect1(), MyAspect2())
```

Note: The order of aspects matters; the aspects are applied from left-to-right, with left being the most deeply nested (the innermost)

Also note that the weaver expects *instances* of aspect classes.

Now, each class which may be affected by aspects, should declare its meta-class to be the aspect weaver. For example:

Module *someclass1.py*:

```
from metaclassconfig import MyMetaClass

class SomeClass1(object):
    __metaclass__ = MyMetaClass
    def some_method(self):
        # ...
        pass
    def some_other_method(self):
        # ...
        pass
```

Here, `MyAspect1` will be applied only to *SomeClass1.some_method*, while both `MyAspect1` and `MyAspect2` will be applied, in that order to both methods of `SomeClass1`

class `libutilitaspy.aspects.core.`**Aspect**

An aspect class defines a *pointcut* (the set of *joinpoints* where the aspect is to be applied, and *advice* methods *before* and *after* to be executed before (resp. after) the joinpoint.

The pointcut is specified by defining the following class attributes:

- *classes*
- *methods*

In a future version, the following will be supported as well:

- *get*
- *set*

Each of these class attributes is a regular expression (see http://docs.python.org/library/re.html). Together they determine the classes and methods where the aspect is to be applied.

Note: This is an abstract class, meant to be subclassed by concrete aspects.

---

before (*klass*, *method*, *obj*, **args*, ***kwargs*)

Performs some actions before the execution of the method.

Parameters

- **klass** – The class of the joinpoint.
- **method** – The method of the joinpoint (i.e. the method executed).
- **obj** – The instance to which the method was applied.
- **args** – The (positional) arguments to the method.
- **kwargs** – The keyword arguments to the method.

Returns

None: Indicates that execution proceeds normally and control is passed to the method.

value: Any value. Indicates that the wrapper does not call the method, and overrides it by returning value instead.

after (*klass*, *method*, *obj*, *retval*, *exc_type*, *exc_val*, *traceback*)

Performs some actions after the execution of the method.

Parameters

- **klass** – The class of the joinpoint.
- **method** – The method of the joinpoint (i.e. the method executed).
- **obj** – The instance to which the method was applied.
- **retval** – The value which was returned by the method, if it returned normally, or None if it raised an exception.
- **exc_type** – The type of exception if one was raised, or None if the method returned normally.
- **exc_val** – The exception instance if one was raised, or None if the method returned normally.
- **traceback** – The traceback of the exception if one was raised, or None if the method returned normally.

Returns

None: Indicates that the wrapper should return retval when the method returns normally or raise exc_val when the method raises an exception.

value: Any value. Indicates that the wrapper overrides the method's return value or exceptions, and returns value instead.

libutilitaspy.aspects.core.**WeaverMetaClassFactory**(**aspects*)
This factory function produces a meta-class which weaves the given aspects in all classes which are instances of the meta-class.

Parameters **aspects** – a sequence of `Aspect` instances

Returns An aspect weaver meta-class (to be assigned to the *__metaclass__* attribute of the client classes.

libutilitaspy.aspects.core.**make_aspect_from_generator**(*generator*)
Makes an `Aspect` class from a given generator. The idea is that whatever comes in the generator function before a *yield* becomes the *before* part of the aspect, and whatever comes after the *yield* becomes the *after* part of the aspect.

Parameters **generator** (*GeneratorType*) – some generator (typycally a generator function).

Returns An aspect with before and after methods

Return type `Aspect`

2.4.2 libutilitaspy.aspects.logger

This module implements a general Logger aspect for logging method calls.

@author Ernesto Posse

2.4.3 libutilitaspy.aspects.memoizer

This module implements a general Memoizer aspect for memoizing (caching) method results.

@author Ernesto Posse

class libutilitaspy.aspects.memoizer.**Memoizer**
A memoizer aspect decorates methods with actions to remember previously computed values for the methods and the given arguments to avoid recomputing them.

Warning: it is not thread-safe, therefore if a method is executed in multiple threads, the cached results may be inconsistent.

before(*klass*, *method*, *obj*, **args*, ***kwargs*)
Try to return the previously computed value for the method, the given object and given arguments, if it has already been computed and stored in the cache table. If it has not been computed, create a new entry in the cache table for the method, the given object and given arguments.

after(*klass*, *method*, *obj*, *retval*, *exc_type*, *exc_val*, *traceback*)
Store the return value of the method in the cache table.

2.5 testing package

This package contains modules that extend unittest, for unit testing and regression testing.

`testing.extended_test_cases`	Author: Ernesto Posse
`testing.standard_tests`	Test cases for common languages
`testing.runtests`	Regression testing framework

2.5.1 libutilitaspy.testing.extended_test_cases

Author: Ernesto Posse Created on: Nov 4, 2010 Description:

> This module contains an extension to the TestCase class from the standard unittest module to support an 'assertDoesNotRaise' method.

2.5.2 libutilitaspy.testing.standard_tests

Test cases for common languages

2.5.3 libutilitaspy.testing.runtests

Regression testing framework Author: Ernesto Posse

Description: This script runs a collection of unit tests in a given directory tree. It's purpose is to easily perform regression testing.

It collects all test cases (subclasses of unittest.TestCase) found in the directory tree in every python file whose name satisfies a given pattern, and only in directories which satisfy the given pattern under the given base directory.

Usage: runtests [options] [base_dir]

Options:

-h, --help show this help message and exit

-f FILE_NAME_PATTERN, --files=FILE_NAME_PATTERN Specify the files to include by name pattern (a regular expression). The default is 'test[0-9]*.py$'

-d DIR_NAME_PATTERN, --dirs=DIR_NAME_PATTERN Specify the directories to include by name pattern (a regular expression). The default is '.+$'

-p ADDITIONAL_PATHS, --path=ADDITIONAL_PATHS Specify additional paths to include in the PYTHONPATH. The default is '.'

-v VERBOSITY, --verbosity=VERBOSITY Specify the verbosity of the test run. The default is '2'

libutilitaspy.testing.runtests.**get_cmd_line_opts**()
Returns the result of parsing the command-line options.

libutilitaspy.testing.runtests.**get_module_names**(*base_dir='/home/eposse/Dropbox/Projects/libutilitaspy/doc'*, *file_name_pattern='test[0-9]*.py$'*, *dir_name_pattern='.+$'*)
Returns the list of qualified module names for modules that meet the criteria that: 1) are defined inside a package within the base_dir directory 2) their file name matches the given pattern 3) their containing directory is a package that matches the given pattern.

2.6 general package

This package contains modules that implement general utilities.

general.decorators	This module contains some generally useful function and method decorators.
general.infinity	Module: infinity
general.utils	

2.6.1 libutilitaspy.general.decorators

This module contains some generally useful function and method decorators.

libutilitaspy.general.decorators.**sig**(*params*, *ret*)

Decorator for functions: decorates a function with parameter and return types. Type checking is performed dynamically.

Typical use:

```
@sig([type_1, ..., type_n], type_ret)
def func(param_1, ..., param_n):
    ...
    return ...
```

When invoking *func(arg_1, ..., arg_n)*, this decorator checks that *arg_i* is an instance of *type_i*, and that the returned value is an instance of *type_ret*.

Parameters

- **params** (*type list*) – list of parameter's types
- **ret** (*type*) – return type

Returns A "typecheck" function which accepts as parameter a function f and returns the decorated function.

Return type function decorator (function -> function)

libutilitaspy.general.decorators.**msig**(*params*, *ret*)

Decorator for methods: decorates a method with parameter and return types. Type checking is performed dynamically.

This is just like the sig decorator, but ignores the first argument.

Typical use:

```
@msig([type_1, ..., type_n], type_ret)
def meth(self, param_1, ..., param_n):
    ...
    return ...
```

When invoking *func(arg_1, ..., arg_n)*, this decorator checks that *arg_i* is an instance of *type_i*, and that the returned value is an instance of *type_ret*.

Parameters

- **params** (*type list*) – list of parameter's types
- **ret** (*type*) – return type

Returns A "typecheck" function which accepts as parameter a function f and returns the decorated function.

Return type method decorator (method -> method)

2.6.2 libutilitaspy.general.infinity

Module: infinity Author: Ernesto Posse Created on: Jul 6, 2007 Last modified: Aug 6, 2010

Description: This implements a class to represent infinity and perform arithmetic operations (e.g. num + inf = inf)

2.6.3 libutilitaspy.general.utils

libutilitaspy.general.utils.**fit**(*s*, *n*, *filler=' '*, *lpad=0*, *rpad=0*)
This is equivalent to filler * lpad + s.ljust(n,filler)[:n] + filler * rpad

libutilitaspy.general.utils.**make_hashable**(*val*)
Returns a hashable value of the given value. Warning: it will not terminate on recursive data structures.

libutilitaspy.general.utils.**get_attributes**(*obj*)
Returns the list of 'visible' attributes of obj.

CHAPTER

THREE

INDICES AND TABLES

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

BIBLIOGRAPHY

[GoF94] 5. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of reusable Object-Oriented Software. 1994

PYTHON MODULE INDEX

l

INDEX

M

N

O

P

R

S

T

U

W