|  |  |  |
| --- | --- | --- |
| 252233-FITOSMART: PLATAFORMA TECNOLÓGICA DE FITOMONITORIZACIÓN DE CULTIVO HIDROPÓNICO UTILIZANDO CÓMPUTO SENSIBLE AL CONTEXTO Y TÉCNICAS DE INTELIGENCIA ARTIFICIAL  (Tercera Etapa) | | Programa de Estímulos a la Innovación  2018 |
| **Código Fuente**  **Módulo De Sistema De Plan De Cultivo** |  | |

**Código Fuente**

**main.py**

#!/usr/bin/env python2

## -\*- coding: utf-8 -\*-

import os, sys

try:

import pickle as pickle

except ImportError:

import pickle

try:

import readline, atexit

history\_filename = "cbr\_command\_history"

def save\_history(path=history\_filename):

import readline

readline.write\_history\_file(path)

if os.path.exists(history\_filename):

readline.read\_history\_file(history\_filename)

atexit.register(save\_history)

except ImportError:

pass

case\_filename = "cases.pickle"

def main():

from matcher import Matcher

from interface import Interface

import attribute\_names

if os.path.exists(case\_filename):

with open(case\_filename, "rb") as fp:

ranges,cases = pickle.load(fp)

for k,v in list(ranges.items()):

atrcls = getattr(attribute\_names, k)

atrcls.\_range = v

else:

print("Warning: No cases found (looking in '%s')." % case\_filename)

cases = []

matcher = Matcher(cases)

interface = Interface(matcher)

interface.cmdloop()

if \_\_name\_\_ == "\_\_main\_\_":

try:

main()

except RuntimeError as e:

sys.stderr.write("Fatal error occurred: %s\n" % e)

sys.exit(1)

**case.py**

## -\*- coding: utf-8 -\*-

\_\_all\_\_ = ['Case']

from attributes import BaseAttribute

import attribute\_names

class Case(dict):

"""Class to represent a case.

This is basically a dictionary that only accepts keys that have an

attribute class defined in attributes.attribute\_names, and

converts its keys into Attribute classes.

Apart from the normal dictionary methods, similarity() and adapt()

are defined, to respectively compare cases and adapt one case to

another."""

def \_\_init\_\_(self, values={}, \*\*kwargs):

"""Constructor populates the case with the dictionary values

and/or the kwargs."""

for key,value in list(values.items()) + list(kwargs.items()):

self[key] = value

def \_\_setitem\_\_(self, name, value):

"""Overridden \_\_setitem\_\_ to turn attributes into attribute

classes before setting them (and raising an error if an

appropriate attribute object cannot be found).

If an Attribute instance is assigned to a key, it is set as

the key directly. Otherwise, a new Attribute object is always

created for a value. The fact that attributes are never

modified makes it safe to share them between classes."""

if isinstance(value, BaseAttribute):

super(Case, self).\_\_setitem\_\_(name,value)

else:

if not hasattr(attribute\_names, name):

raise KeyError("Unable to process attribute name: %s" % name)

super(Case, self).\_\_setitem\_\_(name,getattr(attribute\_names, name)(value))

def \_\_repr\_\_(self):

return "<Case: %s>" % (", ".join(map(repr, list(self.values()))))

def similarity(self, other):

"""Compute total similarity between cases. Total similarity is

calculated as the sum of the similarities for individual

attributes, normalised to the sum of all attribute weights."""

total\_weight = 0.0

total\_similarity = 0.0

for attr in list(self.values()):

if attr.matching:

try:

total\_similarity += attr.similarity(other[attr.name])

total\_weight += attr.weight

except KeyError:

# Happens if other does not have an attribute of

# this name. This is interpreted as a 0 match.

total\_weight += attr.weight

if total\_weight == 0.0:

return 0.0

return total\_similarity / total\_weight

def adapt(self, other):

"""Adapt this case to fit other case.

This is done by combining all attribute adaptations and

applying this final adaptation to all attributes that are set

to be adjusted by adaptation.

Returns a final new case."""

total\_adapt = 1.0

new\_case = Case()

# First pass: Compute adaptation level from all adaptable

# attributes that exists in both case objects.

for attr in list(self.values()):

if attr.adaptable and attr.name in other:

total\_adapt \*= attr.adapt\_distance(other[attr.name])

# Second pass: Copy all attributes into the new object; for

# adaptable attributes, use the values from other, for

# adjustable attributes, adjust them by the adapt value

for attr in list(self.values()):

if attr.adaptable and attr.name in other:

new\_case[attr.name] = other[attr.name]

elif attr.adjustable:

new\_case[attr.name] = attr.adjusted(total\_adapt)

else:

new\_case[attr.name] = attr

return new\_case

**attributes.py**

## -\*- coding: utf-8 -\*-

from abc import ABCMeta, abstractmethod, abstractproperty

from tree import Tree

from util import key\_name

class BaseAttribute(object, metaclass=ABCMeta):

"""Base class that Attribute inherits from. Specifies the

interface that Attribute classes must conform to. Attribute

contains default implementations of all interface methods."""

@abstractproperty

def adaptable(self):

"""Whether or not this attribute can be adapted to form a new

case."""

return self.\_adaptable

@abstractproperty

def adjustable(self):

"""Whether or not this attribute should be adjusted from the

resulting magnitude of the adaptation of other attributes

(e.g. adjust price based on adjustment for number of persons)"""

return self.\_adjustable

@abstractproperty

def matching(self):

"""Is this attribute used in matching cases to each other?"""

return self.\_matching

@abstractproperty

def name(self):

"""Attribute name"""

return self.\_name

@abstractproperty

def value(self):

"""Attribute name"""

return self.\_value

@abstractproperty

def weight(self):

"""Weight for this attribute"""

return self.\_weight

@abstractmethod

def similarity(self, other):

"""Similarity metric between 0 and the selected weight."""

pass

@abstractmethod

def adapt\_distance(self, other):

"""Return the adaptation distance, which is a positive or

negative value in the range [0-1] signifying how large an

adaptation is required to turn this attribute value into the

other one."""

pass

@abstractmethod

def adjusted(self, other):

"""Adjust this attribute by a percentage. Return new attribute

with the adjusted value."""

pass

class Attribute(BaseAttribute):

"""Base Attribute class, providing an attribute with a name and a

value, defining equality on these, and providing the similarity

distance (which by default always matches)."""

\_adaptable = False

@property

def adaptable(self):

"""Whether or not this attribute can be adapted to form a new

case."""

return self.\_adaptable

\_adjustable = False

@property

def adjustable(self):

"""Whether or not this attribute should be adjusted from the

resulting magnitude of the adaptation of other attributes

(e.g. adjust price based on adjustment for number of persons)"""

return self.\_adjustable

\_matching = True

@property

def matching(self):

"""Is this attribute used in matching cases to each other?"""

if hasattr(self, "\_matching\_set"):

return self.\_matching\_set

return self.\_matching

@matching.setter

def matching(self,value):

self.\_matching\_set = value

def scale(self, value, input\_vals=None):

"""Scale for normalising similarity values.

Supports specifying a \_scale attribute or a \_range attribute,

from which the scale is then calculated.

If a range is specified, it is extended if the scaled

input\_values are supplied and are outside the range."""

if hasattr(self, '\_range'):

if input\_vals:

min\_val = min([self.\_range[0]] + input\_vals)

max\_val = max([self.\_range[1]] + input\_vals)

return value/(max\_val-min\_val)

else:

return value/(range[1]-range[0])

if hasattr(self, '\_scale'):

return value/self.\_scale

return value

@property

def name(self):

"""Attribute name"""

return self.\_\_class\_\_.\_\_name\_\_

@property

def value(self):

"""Attribute value"""

return self.\_value

@value.setter

def value(self,value):

if type(value) == type(self):

self.\_value = value.\_value

else:

self.\_set\_value(value)

def \_set\_value(self, value):

"Setter for value - to be overridden in subclasses"

self.\_value = value

\_weight = 1.0

@property

def weight(self):

"""Weight for this attribute"""

return self.\_weight

def \_\_init\_\_(self, value=None):

self.value = value

def similarity(self, other):

"""Similarity metric between 0 and the selected weight. By

default attributes with the same name are always equal."""

if self.name == other.name:

return self.weight

else:

return 0.0

def adapt\_distance(self, other):

"""Return the adaptation distance, which is a positive or

negative value in the range [0-1] signifying how large an

adaptation is required to turn this attribute value into the

other one."""

raise NotImplementedError

def adjusted(self, value):

"""Adjust this attribute by a percentage. Return new attribute

with the adjusted value."""

raise NotImplementedError

def \_\_eq\_\_(self, other):

"""Equality is on all attributes"""

if isinstance(other, BaseAttribute):

return self.name == other.name and self.value == other.value and self.weight == other.weight

else:

return self.value == other

def \_\_ne\_\_(self,other):

return not self.\_\_eq\_\_(other)

def \_\_repr\_\_(self):

return "<Attr %s: %s>" % (self.name, self.value)

def \_\_str\_\_(self):

"""The string representation of an attribute is its value by default"""

return str(self.value)

class ExactMatch(Attribute):

"""Exact matching attribute, that provides a full (i.e. weight

match) on the same value, and zero similarity otherwise."""

def similarity(self, other):

if self.name == other.name and self.value == other.value:

return self.weight

else:

return 0.0

class CaseLessMatch(Attribute):

"""Case-insensitive match on attribute value."""

def similarity(self, other):

if self.value.lower() == other.value.lower():

return self.weight

else:

return 0.0

class Numeric(Attribute):

"""Attribute with positive numeric values."""

def \_set\_value(self,value):

if type(value) == type(self):

self.\_value = value.\_value

else:

try:

val = int(value)

except ValueError:

raise ValueError("Unrecognised value for %s: '%s'." % (self.name, value))

else:

if val < 1:

raise ValueError("Only positive values permitted.")

self.\_value = val

class LinearMatch(Numeric):

"""Matches linearly on a numeric attribute value."""

\_scale = 1.0

def similarity(self, other):

"""Linear similarity metric - absolute value of numeric

difference, scaled by self.scale."""

return self.weight\*(1.0-self.scale(abs(self.value-other.value), [self.value, other.value]))

class NumericAdapt(Numeric):

"""Exact match, but allow numeric adaptation based on this

attribute."""

\_adaptable = True

def adapt\_distance(self, other):

"""Return the adaptation distance, i.e. the ratio of the other

value to the current value. This allows several cumulative

adaptations to be combined by simple multiplication.

Straight-forward numeric fraction of difference in relation to

the current value."""

return float(other.value)/self.value

class LinearAdjust(Attribute):

"""Linear numeric adjustment of value."""

\_adjustable = True

def adjusted(self, value):

return self.\_\_class\_\_(self.value \* value)

class LessIsPerfect(LinearMatch):

"""A 'Less is perfect' match, which is a linear match except when

the other value is less than this one, in which case it is a

perfect match."""

def similarity(self,other):

if other.value < self.value:

return self.weight

return LinearMatch.similarity(self,other)

class MoreIsPerfect(LinearMatch):

"""A 'Less is perfect' match, which is a linear match except when

the other value is less than this one, in which case it is a

perfect match."""

def similarity(self,other):

if other.value > self.value:

return self.weight

return LinearMatch.similarity(self,other)

class TableMatch(Attribute):

"""Table matching, by comparing values to a predefined table

(nested dictionaries) to get a similarity measure."""

\_match\_table = {}

def similarity(self, other):

return self.\_match\_table[self.value][other.value] \* self.weight

def \_set\_value(self, value):

try:

self.\_value = key\_name(value, self.\_match\_table)

except KeyError:

raise ValueError("Unrecognised value for %s: '%s'." % (self.name, value))

class TreeMatch(Attribute):

"""Tree matching, by finding the nearest common ancestor between two values."""

\_match\_tree = Tree(["root", 0.0, []])

def similarity(self, other):

if self.value == other.value:

return self.weight

return self.\_match\_tree.find\_common\_value([self.value, other.value]) \* self.weight

def \_set\_value(self, value):

if self.\_match\_tree.find\_path(value) is None:

raise ValueError("Unrecognised value for %s: '%s'." % (self.name, value))

self.\_value = value

**place.py**

## -\*- coding: utf-8 -\*-

import os, atexit

try:

import pickle as pickle

except ImportError:

import pickle

try:

from geopy import geocoders, distance

except ImportError:

raise RuntimeError("Could not find geopy library. See http://code.google.com/p/geopy/.")

try:

geocoder = geocoders.Google(domain="maps.google.co.uk")

except AttributeError:

geocoder = geocoders.GoogleV3(domain="maps.google.co.uk")

location\_cache\_filename = "location\_cache.pickle"

def save\_location\_cache(filename=location\_cache\_filename):

import place

with open(filename, "wb") as fp:

pickle.dump(place.location\_cache, fp, -1)

atexit.register(save\_location\_cache)

location\_cache = {}

if os.path.exists(location\_cache\_filename):

with open(location\_cache\_filename, "rb") as fp:

try:

location\_cache = pickle.load(fp, encoding="utf-8")

except pickle.UnpicklingError:

pass

# Table of replacement keys to get the right place results on a google

# search. Source: Wikipedia :)

correction\_table = {"fano": "fanø",

"czechia": "czech republic",

"erz gebirge": "erzgebirge",

"turkish aegean sea": "aegean sea",

"riviera": "french riviera",

"turkish riviera": "istanbul", # close enough

"costa blanca": "costa blance, spain",

"teneriffe": "tenerife",

"salzberger land": "salzburg",

"costa brava": "costa brava, spain",

"atlantic": "bordeaux", # It's by the Atlantic, in France

"algarve": "algarve, portugal",

}

class Place(object):

def \_\_init\_\_(self, name):

self.name = name

key = name.lower()

if key in correction\_table:

key = correction\_table[key]

if not key in location\_cache:

try:

search\_value = list(geocoder.geocode(key, exactly\_one = False))

location\_cache[key] = search\_value[0]

except:

raise ValueError("Unable to find location: '%s'" % name)

location\_cache[key] = (None,None)

self.place\_name, self.coords = location\_cache[key]

def latitudal\_distance(self, other):

"""Latitudal distance between two places."""

if self.coords is None or other.coords is None:

return 0.0

return abs(self.coords[0]-other.coords[0])

def distance(self, other):

return distance.distance(self.coords, other.coords).km

def \_\_repr\_\_(self):

return "<Place: %s>" % repr(self.place\_name)

**interface.py**

## -\*- coding: utf-8 -\*-

\_\_all\_\_ = ['Interface']

import re, inspect, sys, cmd

from console import Console

from case import Case

from table\_printer import print\_table

from util import key\_name

from matcher import AdaptationError

import attribute\_names

# Possible attribute names are all classes defined in the attribute\_names module

possible\_attributes = dict(inspect.getmembers(attribute\_names, inspect.isclass))

class Interface(Console):

\_default\_config = {"retrieve": 2,

"adapt": True,

"auto\_run": True,

"auto\_display": True,

"verbose\_results": False}

def \_\_init\_\_(self, matcher):

Console.\_\_init\_\_(self)

self.config = self.\_default\_config

self.prompt = ">> "

self.intro = "Welcome to the CBR system. Type 'help' for a list of commands."

self.matcher = matcher

if not self.matcher.cases:

self.intro += "\nNOTE: Currently no cases loaded (you may want to run parser.py to generate some)!"

self.query = Case()

self.result = []

if not sys.stdin.isatty():

self.prompt = self.intro = ""

self.interactive = False

self.config['auto\_run'] = False

else:

self.interactive = True

def gen\_help(self, method):

"""Generate a help message by removing extra spaces from doc strings"""

if isinstance(method, str):

helpstring = getattr(self.\_\_class\_\_, method).\_\_doc\_\_

else:

helpstring = method.\_\_doc\_\_

return re.sub("\n \*", "\n", helpstring)

def do\_help(self, arg):

if arg in ('status', 'query', 'result', 'config', 'exit'):

Console.do\_help(self, arg)

else:

print("\n".join(['These are the accepted commands.',

'Type help <command> to get help on a specific command.',

'',

'status Show summary of system status.',

'query Manipulate and run query.',

'result Show result of a query.',

'config Set config variables.',

'exit Exit application.']))

def help\_help(self):

print(self.gen\_help("do\_help"), end=' ')

def do\_status(self, arg):

"""Print current status of system (i.e. how many cases loaded etc)."""

print("Currently %d cases loaded." % len(self.matcher.cases))

if self.query:

print("Current query has %d attributes." % len(self.query))

else:

print("No current query.")

if self.result:

print("Result exists.")

else:

print("No result exists.")

def help\_status(self):

print(self.gen\_help("do\_status"))

def do\_query(self, arg):

"""Manipulate the query.

query [show] Show current query.

query reset Reset query to be empty.

query set <attribute> <value> Set query attribute <attribute> to <value>.

query unset <attribute> Unset query attribute <attribute>.

query names [attribute] Show possible attribute names.

query run Run the current query.

By default, the query is automatically run when changed, and

the result is automatically displayed when run. This behaviour

can be changed by setting respectively the 'auto\_run' and

'auto\_display' config parameters."""

if arg in ('', 'show'):

if self.query:

print\_table([self.query], ["Attribute", "Value"])

else:

print("No current query.")

elif arg == "reset":

self.query = Case()

elif arg.startswith('set'):

parts = arg.split(None, 2)

if len(parts) < 3:

print("Usage: query set <attribute> <value>.")

return

arg,key,val = parts

try:

self.query[key\_name(key, possible\_attributes)] = val

if self.config['auto\_run']:

self.do\_query("run")

except KeyError:

print("Invalid attribute name '%s'." % key)

print("Possible attribute names:")

print("\n".join([" "+i for i in sorted(possible\_attributes.keys())]))

except ValueError as e:

print(str(e))

elif arg.startswith('unset'):

parts = arg.split()

if len(parts) < 2:

print("Usage: query unset <attribute>.")

return

arg,key = parts[:2]

try:

key = key\_name(key, possible\_attributes)

del self.query[key]

if self.config['auto\_run']:

self.do\_query("run")

except KeyError:

print("Attribute '%s' not found." % key)

return

elif arg.startswith('names'):

parts = arg.split()

if len(parts) < 2:

print("Possible attributes:")

print\_table([dict([(k,v.\_weight) for (k,v) in list(possible\_attributes.items())]),

dict([(k,v.\_adaptable) for (k,v) in list(possible\_attributes.items())]),

dict([(k,v.\_adjustable) for (k,v) in list(possible\_attributes.items())]),],

["Attribute name", "Weight", "Adaptable", "Adjusted"])

print("\n".join(("Weight is the weight of the attribute for case similarity.",

"",

"Adaptable specifies whether the attribute can be adapted to",

"the query value.",

"",

"Adjustable specifies whether the attribute is adjusted based",

"on the adaptable ones.",

"",

"Run 'query names <attribute>' for help on an attribute.")))

else:

try:

key = key\_name(parts[1], possible\_attributes)

attr = possible\_attributes[key]

print("\n".join(("Attribute : %s" % key,

"Weight : %s" % attr.\_weight,

"Adaptable : %s" % attr.\_adaptable,

"Adjusted : %s" % attr.\_adjustable,

"")))

print(self.gen\_help(attr))

except KeyError:

print("Unrecognised attribute name: %s" % parts[1])

elif arg.startswith('run'):

if not self.query:

print("No query to run.")

return

result = self.matcher.match(self.query, self.config['retrieve'])

if result:

if self.config['adapt']:

try:

result.insert(0, self.matcher.adapt(self.query, result))

except AdaptationError:

pass

self.result = (Case(self.query), result)

if self.config['auto\_display']:

self.do\_result("")

elif self.interactive:

print("Query run successfully. Use the 'result' command to view the result.")

else:

print("no result.")

else:

print("Unrecognised argument. Type 'help query' for help.")

def help\_query(self):

print(self.gen\_help("do\_query"))

def complete\_query(self, text, line, begidx, endidx):

return self.completions(text, line, {'set': sorted(possible\_attributes.keys()),

'names': sorted(possible\_attributes.keys()),

'unset': list(self.query.keys()),

'show': [],

'reset': [],

'run': []})

def do\_result(self, args):

"""Print the current query result.

Prints a table with the query result, each column

corresponding to a result. The query that is printed along

with the result.

If the verbose\_results config parameter is set (default: off),

the similarities for each attribute is shown after the value

in the form (normalised/weighed).

If adaptation is turned on (which is the default; turn it off

with 'config set adapt 0'), an adapted version of the best

result is shown along with the results if adaptation is

possible. Adaptation is possible if any of the parameters of

the query are adaptable, and the query value differs from the

value of the best result.

No adaptation is done if the adapted result is worse (i.e. has

a lower similarity) than the best query match. This can happen

if the adjusted attribute is part of the query.

Note that the query shown in the result can differ from the

current one, if the query has been altered and not rerun (by

default, the query is re-run whenever it is altered, but this

can be changed with the 'auto\_run' parameter."""

if not self.result:

print("No result.")

return

query,result = self.result

header = ["Attribute", "Query"]

results = [query]

add = 1

for i,(sim,res) in enumerate(result):

if sim == 'adapted':

header.append("Adapted result (sim. %.3f)" % query.similarity(res))

add = 0

else:

header.append("Result %d (sim. %.3f)" % (i+add, sim))

if self.config['verbose\_results']:

r = {}

for k,v in list(res.items()):

if k in query:

s = query[k].similarity(v)

w = query[k].weight

else:

s = 1.0

w = 1.0

r[k] = "%s (%.2f/%.2f)" % (v, s/w, s)

results.append(r)

else:

results.append(res)

print\_table(results,header)

def help\_result(self):

print(self.gen\_help("do\_result"))

def do\_config(self, args):

"""View or set configuration variables.

config [show] Show current config.

config set <key> <value> Set <key> to <value>.

Configuration keys:

adapt: Whether or not to adapt the best case if not a perfect match.

auto\_display: Automatically display results after running query.

auto\_run: Automatically run query when it changes.

retrieve: How many cases to retrieve when running queries.

verbose\_results: Show similarities (normalised/weighed) for each attribute."""

if args in ('', 'show'):

print("Current config:")

print\_table([self.config], ['Key', 'Value'])

elif args.startswith('set'):

parts = args.split(None, 2)

if len(parts) < 3:

print("Usage: config set <key> <value>.")

return

key,value = parts[1:3]

if not key in self.config:

print("Unrecognised config key: '%s'" % key)

try:

if type(self.config[key]) in (int, float):

self.config[key] = type(self.config[key])(value)

elif type(self.config[key]) == bool:

if value.lower().strip() in ("1", "t", "y", "yes", "true"):

self.config[key] = True

elif value.lower().strip() in ("0", "f", "n", "no", "false"):

self.config[key] = False

else:

raise ValueError

except ValueError:

print("Invalid type for key %s: '%s'" % (key,value))

else:

print("Unrecognised argument.")

self.help\_config()

def help\_config(self):

print(self.gen\_help('do\_config'))

def complete\_config(self, text, line, begidx, endidx):

return self.completions(text, line, {'show': [],

'set': list(self.config.keys())})

def completions(self, text, line, completions):

parts = line.split(None)

current = []

if len(parts) == 1 or (len(parts) == 2 and not parts[1] in list(completions.keys())):

current = list(completions.keys())

elif ((len(parts) == 2 and not text) or (len(parts) == 3) and text) and parts[1] in completions:

current = completions[parts[1]]

return [i+" " for i in current if i.lower().startswith(text.lower())]

def completenames(self, text, line, begidx, endidx):

completions = ['help', 'query', 'status', 'result', 'config', 'exit']

if text==line:

return [i+" " for i in completions if i.startswith(text)]

return Console.completenames(self, text, line, begidx, endidx)

def default(self, line):

print("Invalid command. Type 'help' for a list of commands.")

def postloop(self):

cmd.Cmd.postloop(self)

if self.interactive:

print("Exiting...")

if \_\_name\_\_ == "\_\_main\_\_":

from matcher import Matcher

interface = Interface(Matcher())

interface.cmdloop()

**matcher.py**

## -\*- coding: utf-8 -\*-

class AdaptationError(RuntimeError):

pass

class Matcher(object):

def \_\_init\_\_(self, cases=[]):

self.cases = cases

def match(self, query, count):

"""Match a query to the case base and return the best matches."""

# Construct a list of tuples (similarity, case) from all cases

# in the case base.

similarities = list(zip(list(map(query.similarity, self.cases)), self.cases))

# Return the count first elements of the sorted list of

# similarities (sorted() sorts on the first element of the

# tuple).

return sorted(similarities, key=lambda x: x[0], reverse=True)[:count]

def adapt(self, query, result):

"""Adapt a result to a query, if possible.

The return value is a tuple ('adapted', case), to conform to

the format of the return values of match()."""

if not result:

raise AdaptationError("Cannot adapt from empty result")

# result is assumed to be the result of a call to match(), so

# get the Case element of the best match (i.e. the first

# element).

sim,best = result[0]

# The adaptable attributes are all those that are marked as

# such, and that differ in value between the query and the

# case.

adaptable = [k for (k,v) in list(query.items()) if v.adaptable and query[k] != best[k]]

if not adaptable:

raise AdaptationError("No adaptable values differ")

adapted = best.adapt(query)

if query.similarity(adapted) < sim:

raise AdaptationError("Adapted result is worse than best match")

return ('adapted', adapted)