wk

(wantedkeywords)

Version : 1.0
Date : 19/08/2006
Author : ¨Patrick Germain Placidoux

SUMMARY

1 INTRODUCTION

Wanted Keyword (wk) is a Python/Jython library for control type handling.

Note: should work in Jython in any situation althougth it still experimental.

2 PREREQUISITES

Oses (every where python works)

Linux, AIX ®, Windows ®

Langages

Python >= 2.3
Jython >= 2.5 (experimental)

3 INSTALL

Download wk (wanted keywords) at www.sourceforge.net

Unzip the file wk_#.##.zip in the directory of your choice.

4 QUICK VIEW

A classic python function signature is:

def myFunct(*args, **keyword)
or
def myFunct(**keyword)

where keywords is a set of key pair attribute/values.

We also find:
def myFunct(attr1=value1, attr1=value1, attr1=value3)

wk library help to assume a strict type control on keywords or on any python dictionary passed to it.

```
1. def myFunct(*args, **keywords):
2.       import wk

3.        p=wk.WK()
4.        p.name={'*type':'str'}
5.        p.age={'*type':'int'}
6.        p.address=None
7.        wk.getK( p, keywords=keywords )

8.        print 'name:', p.name, 'age:', p.age, 'address:', p.address
```

```
>>> myFunct('dahlia', 23)
name: dahlia age: 23 address: None
```

Quick:This will check that the value for key name within keywords is a StringType
and keywords ['age'] is an IntType.

Detail:
2. p is a new instance of a WantedKeyword class (WK is a shortcut for WantedKeyword).
3. The literal {'*type':wk.str} is the wkDefinition for name.
A wkDefinition is structured dictionary with a set of specific keys called wkKeys.
This wkDefinition says that the expected type for keywords ['name '] is StringType.

4. The wkDefinition: {'*type':wk.int} for age says that the expected type for age is IntType.
7. Calling the function getK on module wk does the real job (getK is a shortcut for getKeywords).
Two parameters are passed, the p instance and the dictionary to be checked: keywords.
8. Print the resulting values.

Note that on start the p instance attributes: p.name and p.age are feed with wkDefinitions,
and on return from wk.getK they are feed with the keywords **real** values.

Terms: inside a wkDefinition
_ Anything starting with an "*" is called a wkKey.
So *type and *required are wkKeys.
wk supports an arbitrary set of wkKeys.
_ Anything that feeds the wkKey: *type is called a wkType.
wk supports an arbitrary set of wkTypes.

- **The Required wkKey**

For the exemple above, a value is not required as long as the wkKey *required is not used.

So calling:
>>> myFunct()
Would return with no exception:
name: None age: None address: None

To require the parameter name, the wkKey: *required must be used like this

```
4.  p.name={'*type':'str', '*required': True}
```

- **Default value**

Default values can be set via the wkKey: *value

```
5. p.age={'*type':'int', '*required': True, '*value':25}
```

So calling:

```
1. def myFunct(*args, **keywords):
2.         import wk

3.         p=wk.WK()
4.         p.name={'*type':'str', '*required': True}
5.         p.age={'*type':'int', '*required': True, '*value':25}
6.         p.address='Unknown address'
7.         wk.getK( p, keywords=keywords )

8.         print 'name:', p.name, 'age:', p.age, 'address:', p.address
```

>>> myFunct(name='dahlia')
Would return:
name: dahlia age: 25 address: Unknown address

Even we didn't provide parameter for age, its returns the default value.

Note:
_ when default is used with *value, *required parameter is not really exploited.

Default value check:

Default value are aslo exposed to wkDefinition check
Example:

```
p=wk.WK()
p.age={'*type':int, '*value': 'rose'}
wk.getK( p, keywords={'age': None} )
```

This would throw a wkExcpetion because 'rose' is string while
the expected *type is int.

- **Arbitrary value**

But why does print show the value 'Unknown address' for the p instance attribute address even we did not have defined any wkDefinition for it ?

In fact it is allowed to delcare a parameter directly with an arbitrary value without any wkdefinition.

In that case a default wkdefinition of *type: **'str'** is setup for it, and its arbitrary value is take as a default value for wkKey: *value.
So

6. p.address='Unknown address'

is the same as:

6. p.address={'*type': 'str', '*value': 'Unknown address' }

- **Supported wktypes**

The several types allowed within the wkKey *type are:
str, int, long, foat, tuple, list, dict, bool, wkDef, date, ts, xml, color, url

Note: date, ts (TimeStamp) , xml, color, url : are supported but experimental.

The weird wkDef : will check that the value for the attribnute is a wkDefinition
(for advanced purposes only).

- **Supported wkKeys**

The supported wkKeys list is:
***type, *value, *required, *lt, *gt, *eq, *le, *ge, *ne, *checkIn, *checkXIn, *ltype, *dtype, *withEval, *withCoolTyping, *between, *maxLen, *regex, *force_str.**

Almost any wkKeys can be combined together.

See the chapter wkKeys to learn how to use each of them.

5 WKKEYS

The supported wkKeys are:
*type, *value, *required, *lt, *gt, *eq, *le, *ge, *ne, *checkIn, *checkXIn, *ltype, *dtype, *withEval, *withCoolTyping, *between, *maxLen, *regex, *force_str.

This chapter details each of them.

5.1 BASIC WKKEYS

5.1.1 *type

syntax : '*type': <TYPE>
TYPE must be in :str, int, long, foat, tuple, list, dict, bool, wkDef, date, ts, xml, color, url

sample:
```
>>> p=wk.WK()
>>> p.age={'*type':'int'}
>>> wk.getK( p, keywords={'age': 30} )
```

5.1.2 *value

syntax : '*value': <VALUE>
VALUE is a default value, if no value is provided for the keyword entry.

sample:
```
>>> p=wk.WK()
>>> p.age={'*value:25}
>>> wk.getK( p, keywords={'age': None} )

print 'age:', p.age
age: None
```

Because the age entry value is None, the resulting value for p.age is None.

5.1.3 *required

syntax : '*required': <True/False>
Not providing the wkKey: *required, is equivalent to *required: False.

sample:
```
>>> p=wk.WK()
>>> p.age={'*required': True}
>>> wk.getK( p, keywords={'age': None} )
```

Would througth a wkException because age is None (not ptrovided).

5.1.4 *lt, *gt, *eq, *le, *ge, *ne

*lt (leater), *gt (greater), *eq (equal), *le (leater or equal), *ge (greater or equal) and *ne (not equal) support the sames syntax.

syntax : '*lt': <VALUE>
Value can be of any type.

sample:
```
>>> p=wk.WK()
>>> p.age={'*type':'int', '*lt':30}
>>> wk.getK( p, keywords={'age': 31} )
```

Would througth a wkException because age 31 is not leater than 30.

Be aware to use and adjust *type with the type you expect.
Dont forget that if no type is provided the entry is converted to the default type str.

A common error is :
```
>>> p=wk.WK()
>>> p.age={ '*eq':30}
>>> wk.getK( p, keywords={'age': 30} )
```

And this wont work because *type not beiing provided, entry is converted to the default type: str.
Then afterward it is compared to an int :
'30' == 30 is false.

So use type to avoid this. This works:
```
>>> p=wk.WK()
>>> p.age={ '*type': int, '*eq':30}
>>> wk.getK( p, keywords={'age': 30} )
```

5.1.5 *checkIn

syntax : '*checkIn': <LIST/TUPLE>
Checks that the value for the entry is in the list.
Note: that any value of any type can be in this list.

sample:
```
>>> p=wk.WK()
>>> p.age={'*checkIn': (25, 30), '*type': 'int'}
>>> wk.getK( p, keywords={'age': 27} )
```

Wont work and a wkException is thrown, because 27 is not into the list.

Be aware to use and adjust *type with the type you expect.
Dont forget that if no type is provided the entry is converted to the default type str.

5.1.6 *checkXIn

checkXIn is the same as checkIn, except that it works with '*type':list.

syntax : '*checkXIn': <LIST/TUPLE>
Checks that each item of the value for the entry is in the list.
Note: that any value of any type can be in this list.

sample:
```
>>> p=wk.WK()
>>> p.age={'*type': 'list', '*checkXIn': (25, 30, 45)}
>>> wk.getK( p, keywords={'ages': (25, 30) )
```

Will work because instead of checking if the tuple : (25, 30)
is in the tuple : (25, 30, 45) like checkIn would.
checkXIn will checks each element of the entry: (25, 30) for its presence into the mandated tuple: (25, 30, 45).

5.1.7 *between

syntax : '*between': <TUPLE/LIST>
The value for between is a 2 element long tuple or list.
The checked value must be greater or equal to the left element,
and leater or equal to the rigth element.

sample:
```
>>> p=wk.WK()
>>> p.age={'*type': 'int', '*between: (45, 75)}
>>> wk.getK( p, keywords={'age': 45} )
```

Will work.

Be aware to use and adjust *type with the type you expect.
Dont forget that if no type is provided the entry is converted to the default type str.

5.1.8 *maxLen

syntax : '*maxLen: <INT>
The value of this wkKey is a guiven lenght.
Allowed *type with this wkkey are: str, list or tuple.
If * type is a str: the length (number of characters) of this str is checked.
If * type is a list/tuple: the length (number of items) of this is list/tuple checked.

sample:
```
>>> p=wk.WK()
>>> p.name ={'*maxLen': 8}
>>> wk.getK( p, keywords={'name': 'palomino'} )
```

This will work because the length of the word: palomino is eight.

5.1.9 *regex

syntax : '*regex': <RE_PATTERN>
A regular expression search is ran on the entry (must be a string),
with this pattern.

sample:
```
>>> p=wk.WK()
>>> p.name ={'*regex': '(?<=palo).+'}
>>> wk.getK( p, keywords={'name': 'palomino'} )

>>> print 'name:', p.name
name : mino
```

Will return mino.

5.2 ADVANCED IMBRICATED WKKEYS

5.2.1 *ltype

syntax : '*ltype': <wkDef>
*ltype is always combined with '*type':'list'.
The *ltype **value** is any **wkDefinition**.

sample:
The simplest list check is:
```
>>> p=wk.WK()
>>> p.figures={'*type':'list'}
>>> wk.getK( wantedKeywords=p, keywords={'figures ': [11, 14]} )
```
This works because [11, 14] is a list (please note that *list accepts a list or a tuple).

A deeper control of the figures list could be appliying a dedicaced
wk definition, to each occurence of the list, this is done by ltype:

```
>>> p=wk.WK()
>>> p.figures={  '*type': 'list', '*ltype': {'*type': 'int',  '*between': (10, 15)} }
>>> wk.getK( wantedKeywords=p, keywords={'figures ': [11, 14]} )
```
And this works because 11 is between 10-15 and 14 as well.

This another sample is a *tlist representation of the wkKey * checkXIn.

```
>>> p=wk.WK()
>>> p.animals={  '*type': 'list', '*ltype': {'*checkIn':('dog', 'fish', 'bird')} }
>>> wk.getK( wantedKeywords=p, keywords={'animals':['dog', 'fish', 'bird']} )
```

Please note that wk is recursive and an imbricated wkKey can support another imbricated wkKey and so long.

5.2.2 *dtype

syntax : '*dtype': a dictionary of <wkDef>.
*dtype is always combined with '*type':'dict' or '*type':'list'.
The value for each **entry** of the dictionary is a **wkDefinition.**
The names the *dtype entries are the allowed entries for the checked dict instance.

Sample:
The simplest dict check is:
```
p=wk.WK()
p.animals={'*type': 'dict'}
wk.getK( p, keywords={'animals': {'kind':'dog', 'fly':False, 'color':'brown', 'number':2}} )
```

This simply checks that p.animals is a dict.

A deeper control of the animals dict could be appliying a dedicaced
wk definition, to each entry of the dict, this is done by *dtype:

```
>>> p=wk.WK()
>>> p.animals  = { '*type':'dict',
                  '*dtype':{
                        'kind':{'*checkIn':('dog', 'fish', 'bird')},
                        'fly':{'*type':'bool'},
                        'color':{'*type':'str', '*checkIn': ('orange', 'brown', 'yellow', 'pink',  'blue')},
                        'number':{'*type':'int', '*lt':10},
                       }
                 }
>>> wk.getK( p, keywords={'animals': {'kind':'dog', 'fly':False, 'color':'brown', 'number':2}} )
```

Sometimes one needs to work with list of dict,
specifically when working with **databases files, tables or grids.**
To operate a deep checking on a list of dict, just replace dict by list in *type:

```
>>> p=wk.WK()
>>> p.animals  = { '*type':'list',
                  '*dtype':{
                        'kind':{'*checkIn':('dog', 'fish', 'bird')},
                        'fly':{'*type':'bool'},
                        'color':{'*type':'str', '*checkIn': ('orange', 'brown', 'yellow', 'blue')},
                        'number':{'*type':'int', '*lt':10},
                       }
                 }

>>> wk.getK( p, keywords={'animals':
        [
                {'kind':'cat', 'fly':False, 'color':'pink', 'number':3},
                {'kind':'dog', 'fly':False, 'color':'brown', 'number':2},
                {'kind':'seagull', 'fly':True, 'color':'grey', 'number':1}
        ]}
)
```

5.3 MARCHALLING WKKEYS

5.3.1 *withEval

syntax : '*withEval: True/False
If True and the received value is str, this will run a secured eval on the value.

sample:
```
>>> p=wk.WK()
>>> p.test={'*withEval': True, '*type': 'list'}
>>> wk.getK( p, keywords={'test':  "['a', 1, 2, 'b']"} )

>>> print 'test:', p.test, type(p.test)
test: ['a', 1, 2, 'b'], list
```

Note:
The check *type: 'list' passes for the value "['a', 1, 2, 'b']", even if it is a str; because the eval conversion is the first operation ran by getK() before all checks.

5.3.2 *withCoolTyping

syntax : '*withCoolTyping': <CoolTyped_E>
Spoking fast a CoolTyped expression is a python expression (bool, int, tuple, dict, ...) converted to str but with no '"', no'"', no trailing space arround structural character (like (,),[,],{,}:).

A CoolTyped expression is easy serializable to a text file, a DB, a command line or an http stream.
The CoolTyping library comes with the wk library and convertes
python expression to ct expresion and versa.

To learn more please check the CoolTyping (ct) library documention.

More:
From this python dict: mydict={'kind':'cat', 'fly':False, 'color':'pink and blue', 'number':3}
a CoolTyped expression is: ct.undress(myDict) : '{fly:False,color:pink and blue,kind:cat,number:3}'
And mydict==ct.dress(ct.undress(myDict)) is True.

sample:
```
>>> p=wk.WK()
>>> p.animals  = { '*type':'dict',
                  '*dtype':{
                        'kind':{'*checkIn':('dog', 'fish', 'bird')},
                        'fly':{'*type':'bool'},
                        'color':{'*type':'str', '*checkIn': ('orange', 'brown', 'yellow', 'pink',  'blue')},
                        'number':{'*type':'int', '*lt':10},
```

```
                    },
                '*withCoolTyping': True
            }
>>> wk.getK( p, keywords={'animals': '{fly:False,color:brown,kind:dog,number:2}'} )

>>> print 'animals', p.animals , type(p.animals )
animals: {'kind':'dog', 'fly':False, 'color':'brown', 'number':2}, dict
```

Note:
The check *type: 'dict' passes for the value '{fly:False,color:brown,kind:dog,number:2}',
even if it is a str; because the ct conversion is the first operation ran by getK() before all checks.

5.3.3 **force_str*

syntax : '*force_str: True/False
When using converter wkKey like *withEval or *withCoolTyping the original value's type wich is always str at the begining switch to another.
'*force_str, force it to switch back to str after convertion.

sample:

```
>>> p=wk.WK()
>>> p.test={'*withEval': True, '*type': 'list', '*force_str': True}
>>> wk.getK( p, keywords={'test':  "['a', 1, 2, 'b']"} )
```

The previous check will fail, because we force the value to become an str again while testing list with *type: list.

```
>>> p=wk.WK()
>>> p.test={'*withEval': True, '*type': 'str', '*force_str': True}
>>> wk.getK( p, keywords={'test':  "['a', 1, 2, 'b']"} )

>>> print 'test:', p.test, type(p.test)
test: ['a', 1, 2, 'b'], str
```

This check succeed.

5.4 ADVANCED FEATURES

5.4.1 *Accecpted combinations*

When within a list of entries some cannot be combined with anothers or if
they are some combinations allowed and others not,
use the the wk extended parameter acceptedComb.

```
>>> p=wk.WK()
>>> p.name={'*required': True}
>>> p.user={'*type':'str'}
>>> p.password={'*type':'str'}
>>> p.passfile={'*type':'str'}

>>> acceptedComb=(
[  'user', 'password', 'passfile'   ],
[     'X',          'X',            ' ',         ],
[      ' ',           ' ',          'X',        ],
[      ' ',           ' ',           ' ',         ],
)

"""
This chows that:
user!=None, password!=None, passfile=None
user=None, password=None, passfile!=None
user=None, password=None, passfile=None
are allowed.
"""
wk.getKeywords(wantedKeywords=p, keywords={ 'name': 'myname', 'passfile':'myfile'}
, remove=True, acceptedComb=acceptedComb)
```

This works.

Lets take a look to this representation, it is a graphical combination representation:

```
acceptedComb=(
[  'user', 'password', 'passfile'   ],
[     'X',          'X',            ' ',         ],
[      ' ',           ' ',          'X',        ],
[      ' ',           ' ',           ' ',         ],
)
```

Please note that a case is filled etiher by 'X' (x in caplock) or ' ' (a blank space).
AcceptedComb is a tuple (or list) of lists of equal size.
The first entry of this list is the list of the attribute names.
The len of this list represents the number of possibilities +1.

If and exception is found running getK(), this same well formated table appears on the terminal, (under *'allowed combinations are:'*) giving the choice to the user to choose is best combination.

6 TRADEMARKS:

AIX is a registred trademarks of International Business Machines Corporation.

Windows is a registred trademark of Microsoft Corporation

Other names may be trademarks of their respective owners.