KCWI: Script Architecture

Author	Date	Comment
Luca Rizzi	27 May 2016	Original

Background

KCWI inherits the rich collection of instrument scripts developed for the other Keck instruments. As a starting point, KCWI scripts are based on MOSFIRE scripts. As a test case, KCWI is exploring the possibility of adopting the Python programming language replacing the previously adopted csh/tcsh/bash model.

Basic concepts

The instrument package
Most of the scripted observing needs are fulfilled not by command line interface scripts (CLIs), but by functions or classes coded in packages. As an example, the traditional "goi" script is replaced by a "goi" function within an instrument package.

```
def goi(nexp, dark=False):
    <body of the function>
```

This piece of code is part of a KCWI package. Note that the standard way of creating a package in python is to create a top level directory with the name of the package, and a set of .py files inside that directory as modules.
Example:

```
KCWI
    |
    |- __init__.py
    |- Detector.py
    |- Blue.py
```

__init__.py can, in general, be left empty. It is just an indication that this is a package. If you want to be able to say "import KCWI" and have all the available functions, then you might want to add explicit imports to your __init__.py:

```
import Detector
import Blue
```

Please refer to https://docs.python.org/2/tutorial/modules.html for further reading on packages.

The current KCWI package's init file contains:

```
''' Library functions to simplify common actions with KCWI.
.. moduleauthor:: L.Rizzi, K. Lanclos
'''

# Import all subcomponents of this module. It's important
not to
# adjust the namespace until all subcompoents have been
imported;
# otherwise, the subcomponent has to use the revised
namespace.

from . import Blue
from . import Calibration
from . import BlueWrapper
from . import Helper
from . import Log
#from . import Red
#from . import RedWrapper
from . import Version
```

For the moment, do not focus on BlueWrapper or RedWrapper

Users will still want to have access to these functions via command line. To write CLIs, we have decided that these scripts should have the least possible amount of logic. That is because "logic" is a synonym with knowledge: if the script needs to know something that the functions don't already know, then you are coding knowledge in two different places, and that is a failure mode ready to happen.
The basic CLI should consist of only two elements: (1) an argument parser and (2) a call to a coded function from the instrument package (or any other package).

The argument parser is an instance of the standard Python ArgumentParser class described here: https://docs.python.org/2/howto/argparse.html.

Rather than learning the entire set of features, I will describe some basic uses in this document.

The second element is the call to a function.
While not strictly necessary, it is good practice to isolate the executable code within a standard "if" statement:

```
if __name__ == '__main__':
    call function
```

This wrapper guarantees that if the script you are writing is imported rather than invoked, the function is not really executed. This might sound a little obscure but I'll give an example: the facility that produces the automatic web based documentation needs to have access to the arguments that you define, but it should not start moving instrument mechanisms. Such a facility will import the scripts, but because they are imported rather than called from the command line, __name__ will not be __main__ and the function will not run. The argument parser is defined outside of the "if" statement, so it is available to whoever has imported the script.

As an example of the goi CLIs:

```
#! /kdevroot/bin/kpython

import argparse
import KCWI

description = "Take KCWI exposure(s)"

parser = argparse.ArgumentParser(description=description)
parser.add_argument('nexp',help='number of exposures', type=int,
default=1, nargs='?')

parser.add_argument('-dark',help='take Dark exposure',required=False,
default=False,action='store_true')

if __name__ == '__main__':

   args = parser.parse_args()

   KCWI.Detector.goi(nexp = args.nexp, dark=args.dark)
```

Parsing arguments

To create an argument parser, start by instantiating the class:

```
parser = arparse.ArgumentParser("this is my best script")
```

The argument is a description of your script, which will be printed on the screen if the user requires help.

Arguments are created by using add_argument.

The most basic types of arguments are required arguments, optional arguments and flags.

For **required arguments**, also known as positional arguments, the syntax is:

```
parser.add_argument('nexp',help='number of exposures',
type=int)
```

In this case, which is not quite real, we require the user to enter the number of exposures on the command line. If the command is called with no argument, we get an error.

If we want to have a named argument that is **optional**, then we can specify that the required number is unknown:

```
parser.add_argument('nexp',help='number of exposures',
type=int, nargs='?')
```

We can also specify a default value for when the argument is missing.

```
parser.add_argument('nexp',help='number of exposures',
type=int, nargs='?', default=1)
```

For flag arguments, the syntax is the same but the argument must start with a "-"

```
parser.add_argument('-dark',help='take Dark
exposure',required=False,
default=False,action='store_true')
```

In this case this line can be translated as: if –dark is specified, the args.dark variable is set to True (store_true). The argument is not required. If omitted, the default value is False. There is a large amount of other options that can be specified: we can specify the list of allowed values and if options are mutually exclusive, for example.

The help page is always available, and is accessed automatically if the user calls the script with –h , or - - help, option. The resulting description can be changed and enriched if needed.

By default, the argument is passed to a variable that has the same name. In this case, the variables would be args.nexp, and args.dark. This can be changed by setting the dest="myname" parameter in the argument definition.

To access the argument and parse the command line, the incantation is:

```
args = parser.parse_args()
```

and the arguments are now in the variables

```
args.nexp and args.dark
```

Interaction with keywords: the KTL-Python interface

For a complete description of this interface, see http://spg.ucolick.org/KTLPython/.

A basic read operation is performed by setting up a call to a service and then reading the keyword:

```
keyword = ktl.cache ('servicename', 'keywordname')
value = keyword.read ()
```

This call "blocks", and returns the ascii representation of the keyword. You can get the binary format by specifying keyword.read(binary=True). You can also specify a timeout if you don't want to wait forever, and a wait=False if you don't want to block at all.

Other simple operations are write:

```
keyword.write(value)
```

and wait for condition to be true:

```
keyword.wait(timeout=60, value=my_expected_value)
```

If you are planning to use the keyword more than once in your script, then it is more useful to establish a monitor, so that every time the keyword is used, the value is current without having to manually specify a new read operation.
If you add a monitor to the keyword, then you are effectively subscribing to updates, which opens the possibility of callbacks. At the very least, this removes the need to read the keyword explicitly.

```
keyword.monitor()
```

If you want to have access to an entire service, such as lris for example to operate a show keyword command, then the syntax is:

```
lris = ktl.cache('lris')
```

At this point you can see which keywords are available:

```
print lris.keywords()
```

If you have already created a service, keywords can be accessed with an alternate syntax:

```
grism = lris['grism']
```

Note that you can access many subcomponents of a keyword.

```
temp = lris['tempdet'] or
temp = ktl.cache('lris','tempdet')


>>> temp['ascii']
'-100.492477'
>>> temp['bin']
-100.49247741699219
>>> temp['units']
'degC'
>>> temp['error']
>>> temp['name']
'TEMPDET'
>>> temp['range']
>>> temp['servers']
('ldserv',)
>>> temp['timestamp']
1464388740.307186
>>> temp['history']
(HistorySlice 1464388390.511188 TEMPDET: -100.492477417/-100.492477,
HistorySlice 1464388394.911218 TEMPDET: -100.648376465/-100.648376,
HistorySlice 1464388396.004915 TEMPDET: -100.492477417/-100.492477,
HistorySlice 1464388739.197231 TEMPDET: -100.648376465/-100.648376,
HistorySlice 1464388740.307186 TEMPDET: -100.492477417/-100.492477)
>>> temp['type']
'KTL_FLOAT'
```

Callbacks

It is possible to trigger calls to a function for each change of the value of a keyword. That is called a callback.
To show how this works, we can construct a copy of the "cshow" command.

```
# define a callback
def mycallback(keyword):
     print keyword['name']+' '+keyword['ascii']

# start a service
myservice = ktl.Service('servicename')

monitored_keywords = ('keyword1','keyword2','keyword3')
for key in monitored_keywords:
     # instantiate the keyword connection
     keyword = service[name]
     # add the callback
     keyword.callback(mycallback)
     # subscribe to broadcast
     keyword.monitor()

# start an infinite loop
while True:
     pass
```

Each time any of the keyword listed in the array monitored_keywords receives an update, the callback is triggered and the name and value of the keywords are printed. Here the list of keywords is hardcoded, but of course they could fed to the script as arguments in an argument parser.

One can see how we could write a logger for example, that monitors temperatures or pressures, by sending new values to a database when they are generated, rather than at fixed intervals. Alarm handlers could be programmed this way too.

Expressions

Any string that contains a collection of logical expressions can be turned in to a virtual keyword that is referred to as an expression.
For example: let's suppose that to know if a filter has reached the final position, we need to know if the current name (fname) matches the target (ftarg) and if the mechanism has stopped moving (fmove, 1 or 0) without errors (fstatus). Let's suppose these are keywords that are related to a server called mech.

We would first define a string describing the expression.

```
move_successful = "($mech.fname == $mech.ftarg) and
($mech.fmove==0) and ($mech.fstatus=='OK')"
```

We then turn this simple string into a new keyword (expression).

```
move_successful = ktl.Expression(move_successful)
```

I am reusing the same name, but it is not necessary.

So if I ask the question "Has the filter move finished correctly?", the answer is:

```
move_successful.eval()
```

```
and I can do:

if move_successful.eval() is True:

     print "Congratulations"
else:
     print "Something happened"
```

Two important things to notice:

1. There is no need for expressions to use keywords related to the same server. The $mech part says that the following keyword belong to that server, but of course anything can be used, including servers that belong to different instrument or to dcs.
2. Expressions have both a "wait" and a "callback" feature. This means that in the previous case, one could say
   `move_successful.wait(timeout=60)`
   To wait for the filter move to be successful.
   One could also create an expression that verify if the filter is dark and if the CSU is moving, and if the expression becomes true at any time, the callback would be triggered, which could be a warning to the user.