Freva – BDG – Basic Developer Guide (ALPHA VERSION)

November 19, 2016

1 Introduction - Welcome to Freva

In this guide we will explain the basic workflow of developing a plugin for Freva.

1.1 Requirements for Plugin Developer

The Central Evaluation System (CES) in MiKlip has different interfaces for an effizient use of data and tools/plug-ins

0. GENERAL
A stand alone tool must have the ability to be used from different users (think of using cdo). The output must land in an (definable) OUTPUTDIR and the temporary files must be in a (defineable) CACHE. No data is allowed to be written in the directories of the tool! It should be installed at the MiKlip Server and well defined complining definitions and/or other programs needed (cdo, R, ncl, etc.)

1. VERSIONING
in MiKlip we are using GIT for version control and an effective development. We have several informations on how2deal with git in the developer guide. It makes sense to have a central repository in and the CES is one "user" using a running version of each tool in plugfins4freva. When a developer have a new running version, the admin just "git pull" it.

2. PLUG-IN
The plug-in framework brings different tools in a common way into the CES. The standalone tool gets a python plugin file ".py" which enables the connection to the CES. There is a how2plugin section with examples and tips. Every tool has different requirements. The plugged-in tools offer a variety of solutions for different tasks. See the ".py" files in the tool section plugins4freva

3. DATA RETRIEVAL via –databrowser
MiKlip decided to work with the CMOR standard. For a common usage of that standard, INTEGRATION developed the searching tool "freva –databrowser". Using this,

enables the developer to offer its userbase a variety of experiments and makes sure all coming experiments in MiKlip are usable with this tool. Because when the directory of the data is changing (what happens more often than you think), the tool developers doesn't need to take care of that.

4. OUTPUT NETCDF
Having output in NETCDF not in GRIB or SRV or ASCII helps to understand what happened with the data. Of course this doesn't work with every output and is depending on the tool.

5. DOCUMENTATION
Of course, the usage of the tool will be clear by just have the "help" coming with plugin-framework (freva –plugin XXXX –help), but a good documentation with examples helps the users to work with it.

6. PRE-PROCESSING via plugged in tools
Re-inventing the wheel doesn't make sense! If wished, you can use the other tools for pre- or postprocessing you evaluation. Maybe "pre" with the leadtimeselektor for using –databrowser in an advanced way including the leadtime ordering of your data and "post" with the movieplotter for plotting your results. Of course this is not necessary with every application.

1.2 Simple Examples

1.2.1 5 Minutes Hands on

The only thing you need to do is to write a python class that inherits from evaluation_system.api.plugin.PluginAbstract and overwrites one method and a couple of arguments.
This is a minimal working plugin, we will describe more interesting examples later on.

```
from evaluation_system.api import plugin, parameters

class MyPlugin(plugin.PluginAbstract):
    tool_developer = {'name':'Max Mustermann', 'email':'max.
       musterman@fu-berlin.de'}
    __short_description__ = "MyPlugin short description (just
       to know what it does)"
    __version__ = (0,0,1)
    __parameters__ =  parameters.ParameterDictionary(parameters
       .Integer(name='solution', default=42))

    def runTool(self, config_dict=None):
        print "MyPlugin", config_dict
```

The plugin itself is not doing much, just printing out the name and the configuration provided when it's being called. But you have a full configurable plugin with a lot of functionality by just inheriting from the abstract class.

So how do we test it? Very, very simply... Here the steps:

Activate the system as described in Using the system

```
module load miklip-ces
```

Create a directory where the plugin will be created

```
mkdir /tmp/myplugin
```

Copy the plugin from above in a file ending in '.py' in the mentioned directory

```
#Use your preferred method of writing the contents to /tmp/
   myplugin/something.py
$ cat /tmp/myplugin/something.py
from evaluation_system.api import plugin, parameters

class MyPlugin(plugin.PluginAbstract):
    __short_description__ = "MyPlugin short description (
       just to know what it does)"
    __version__ = (0,0,1)
    __parameters__ =  parameters.ParameterDictionary(
       parameters.Integer(name='solution', default=42))

    def runTool(self, config_dict=None):
        print "MyPlugin", config_dict

#Setup the environmental variable
   EVALUATION_SYSTEM_PLUGINS=<path>,<package>[:<path>,<
   package>] (that's a colon separated list of comma
   separated pairs which define the location (path) and
   package of the plugin.
#for bash

export EVALUATION_SYSTEM_PLUGINS=/tmp/myplugin,something
```

Test it

```
$ freva --plugin
[...]
MyPlugin: MyPlugin short description (just to know what it
   does)
```

```
$ freva --plugin myplugin
MyPlugin {'solution': 42}

$ freva --plugin myplugin --help
MyPlugin (v0.0.1): MyPlugin short description (just to know
    what it does)
Options:
solution (default: 42)
         No help available.
$ plugin --tool myplugin solution=777
MyPlugin {'solution': 777}
```

That's it!

1.2.2 NCL simple plot

```
NCL simple plot
    Target
    Procedure
    Requirements
    Notes
    Tutorial
        Setting up the environment
        NCL
        Wrapping up the script
        Making a plug-in
        Runing the plugin
    NCLPlot Usage
```

Welcome to the first introduction to the evaluation system (and NCL). We'll try to keep things simple enough while showing you around.

Target

To create a plugin for the evaluation system that just creates a simple plot of a 2D variable.

Procedure

We'll go bottom-up starting from the result we want to achieve and building up the plugin out of it.

Requirements

This tutorial requires you to have a bash shell, so if you are using any other (or don't know what you are using) Just start bash by issuing this at the prompt:

```
bash
```

We'll try to be thorough and assume the user has little knowledge about programming. Though some basic shell knowledge is required. If not, just copy and paste the commands

in here, you don't need to understand everything for getting through the tutorial, though it will certainly help you to locate errors and typos you might have.

Notes

The $ symbol at the beginning of a line is used to mark a command that is given to the bash shell. You don't have to input that character when issuing the same command in your shell.When depicting an interactive shell session it denotes the commands issued to the shell and those lines which does not start with the $ character denote the output from the shell. We might skip the character when there's no expected output from the shell (or if it's not useful), so that you may Copy'n'Paste the whole block. We sometimes number the lines when dissecting a program for didactic reasons. Sadly, this breaks the ability to simply Copy'n'Paste the code. We'll try only to do that on complete programs that you might directly download from this page.

Tutorial Setting up the environment: We'll need a file for testing, grab any suitable for the task. We can use the evaluation system to find one fast.

```
module load project-ces

freva --databrowser --baseline 1  variable=tas | head -n1

/miklip/integration/data4miklip/model/baseline1/output/MPI-M/
    MPI-ESM-LR/asORAoERAa/day/atmos/tas/r1i1p1/tas_day_MPI-ESM-
    LR_asORAoERAa_r1i1p1_19600101-19691231.nc
```

Let's store that in a variable for later use:

```
file=$(find_files --baseline 1  variable=tas | head -n1)
$ echo $file
/miklip/integration/data4miklip/model/baseline1/output/MPI-M/
    MPI-ESM-LR/asORAoERAa/day/atmos/tas/r1i1p1/tas_day_MPI-ESM-
    LR_asORAoERAa_r1i1p1_19600101-19691231.nc
$ export file
```

The last command (export) tells bash to pass that variable to other programs started from this shell (this is one of many methods of passing info to NCL). Ok now let's see what the file has to offer:

```
$ ncdump -h $file | grep ') ;'
    double time(time) ;
    double time_bnds(time, bnds) ;
    double lat(lat) ;
    double lat_bnds(lat, bnds) ;
    double lon(lon) ;
    double lon_bnds(lon, bnds) ;
    float tas(time, lat, lon) ;
```

As expected we have one variable, tas, but check the order in which the values are stored (time, lat, lon) as we will need them later. Now let's start with our bottom-up procedure. We'll start ncl and try to get a plot by using the NCAR tutorial:

```
$ ncl
-bash: ncl: command not found
```

That means we need to load the ncl module, this will also be required when creating the plug-in.

```
$ module load ncl
$ ncl
 Copyright (C) 1995-2012 - All Rights Reserved
 University Corporation for Atmospheric Research
 NCAR Command Language Version 6.1.0-beta
 The use of this software is governed by a License Agreement.
 See http://www.ncl.ucar.edu/ for more details.
ncl 0>
```

NCL

So we'll use NCL to produce a simple plot of a file. Let's use the ncl tutorial here to generate a plot:

```
Now here's the complete session to generate a simple plot out
   of the information in the tutorial (you may Copy'n'Paste it
   into the ncl shell):
\begin{lstlisting}
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
cdf_file = addfile("$NCARG_ROOT/lib/ncarg/data/cdf/contour.cdf
   ","r")
temp = cdf_file->T(0,0,:,:)                    ; temperature
lat  = cdf_file->lat                           ; latitude
lon  = cdf_file->lon                           ; longitude
xwks = gsn_open_wks("x11","gsun02n")           ; Open an X11
   workstation
plot = gsn_contour(xwks,temp,False)            ; Draw a contour plot
   .
```

(Remember to hit enter on the last line too! If not you'll see just an empty window...)

That should have displayed a simple contour plot over some sample data. The ncl shell blocks until you click on the plot (don't ask me why...)
That shows everything is setup as expected. Now let's display the file we selected:

```
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
file_path=getenv("file")                  ; get the value stored
   in the exported variable "file"
```

```
input_file = addfile(file_path,"r")     ; now use that value to
   load the file
temp = input_file->tas(0,:,:)             ; temperature (CMOR
   name tas - surface temparature)
lat  = input_file->lat                    ; latitude
lon  = input_file->lon                    ; longitude
xwks = gsn_open_wks("x11","gsun02n")      ; Open the window
   with the tile "gsun02n" (same as before)
plot = gsn_contour(xwks,temp,False)       ; Draw the contour
   plot again
```

So, the only changes are that in the second line we are retrieving the file name from the environment (remember the "export file" command?), and in the 4th we retrieve a variable named tas instead of T that has only 3 dimensions (2D + time) (remember the the "ncdump -h" command?). So "tas(0,:,:)" means selecting all lat/lon elements of the first time-step. That's what we have plotted.

From the information in the UCAR page we could build a better plot before finishing it up:

```
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
file_path=getenv("file")
input_file = addfile(file_path,"r")
var  = input_file->tas(0,:,:)               ; variable
lat  = input_file->lat
lon  = input_file->lon
resources                  = True           ; Indicate you want
    to set some
                                            ; resources.
resources@cnFillOn         = True           ; Turn on contour
   line fill.
resources@cnMonoLineColor = False           ; Turn off the
   drawing of
                                            ; contours lines in
                                               one color.
resources@tiMainString     = var@standard_name + " (" +
   var@units + ")" ; a title
resources@tiXAxisString    = lon@long_name
resources@tiYAxisString    = lat@long_name
resources@sfXArray         = lon
resources@sfYArray         = lat

xwks = gsn_open_wks("x11","gsun02n")
plot = gsn_contour_map(xwks,var,resources)  ; Draw a map
```

So, we are almost ready, we need to store the result into a file instead of displaying it, pass some parameters (the output directory and file) and extract others, e.g. the variablename is always the first string in the name of the file before the "_" character (Because the files we have follow the DRS standard) so we can use this to our advantage. Another approach for passing values to the NCL program is by using the command line. We'll assume we have a variable called plot_name with the path to the file we want the resulting plot to be stored is already given; the same applies for file_path (just to show a different procedure for passing values around). And last (and definitely least), we'll pack everything in a begin block so we have a proper script.

So this is how it looks like:

```
load "$NCARG_ROOT/lib/ncarg/nclscripts/csm/gsn_code.ncl"
begin
    input_file = addfile(file_path,"r")
    file_name = systemfunc("basename " + file_path)
    tmp = str_split(file_path, "_")
    var_name = tmp(0)
    delete(tmp)

    var  = input_file->$var_name$(0,:,:)        ; first
       timestep of a 2D variable
    lat  = input_file->lat
    lon  = input_file->lon
    resources                  = True           ; Indicate you
       want to set some
                                                ; resources.
    resources@cnFillOn         = True           ; Turn on
       contour line fill.
    resources@cnMonoLineColor = False           ; Turn off the
       drawing of
                                                ; contours
                                                   lines in one
                                                    color.
    resources@tiMainString     = var@standard_name + " (" +
       var@units + ")" ; a title
    resources@tiXAxisString    = lon@long_name
    resources@tiYAxisString    = lat@long_name
    resources@sfXArray         = lon
    resources@sfYArray         = lat

    xwks = gsn_open_wks("eps",plot_name)        ; create an eps
        file
    plot = gsn_contour_map(xwks,var,resources)  ; store the map
```

```
end
```

Store that in a file called plot.ncl.

So how can we test it? Simply by calling ncl in the directory where plot.ncl is located in the following way:

```
ncl plot.ncl file_path=\"$file\" plot_name=\"output\"
```

You should have now a file called output.eps in your current directory which you can view with evince or gs.

You might have notice the strange looking c̈haracters. Ncl requires string values to be always quoted. The shell does use the quotes for something different, so we have to escape them, which basically means telling the shell to pass the " character as is instead of interpreting it in any particular way.
If you want more info on NCL functions: http://www.ncl.ucar.edu/Document/Functions/

Wrapping up the script

One last check... the environment in which we started the previous ncl command had already been setup (with module load ncl). Next time we want to use it might not be the case, or a different ncl version might be loaded, etc. In summary, the program might stop working because the environment is somehow different.

We should assure the environment doesn't change. At least as much as we can.
What we need to do is to clean the environment, check the program doesn't work and set it up again so we are sure we have set it up properly.

Furthermore, we can do that without affecting our current shell by generating a sub-shel in bash by using the () characters. By the way, module purge will remove all modules from the environment. So the test might look like this:

```
$ (module purge && ncl plot.ncl file_path=\"$file\" plot_name
   =\"output\")
bash: ncl: Kommando nicht gefunden.
```

(The && tells bash to issue the next command only if the previous one finished successfully.)

As expected it fails because of the missing ncl command. Now we can test if just adding the ncl module is enough:

```
$ (module purge && module load ncl && ncl plot.ncl file_path=\"
   $file\" plot_name=\"output\")
 Copyright (C) 1995-2012 - All Rights Reserved
 University Corporation for Atmospheric Research
```

```
NCAR Command Language Version 6.1.0
The use of this software is governed by a License Agreement.
See http://www.ncl.ucar.edu/ for more details.
```

Now we are sure it works as intended.

Making a plug-in

Up to now we've seen an introduction to making a script that's usable. This has nothing to do with the system, but plain informatics. Since making a plug-in is so easy, we had to write something to fill up the tutorial :-)

So now we just have to wrap it up in our plug-in. From the information in Developing a plugin, Reference and some python knowledge we (at least I) could infer a very simple plugin:

```
import os
from evaluation_system.api import plugin, parameters

class NCLPlot(plugin.PluginAbstract):
    tool_developer = {'name':'Christopher Kadow', 'email':'
       christopher.kadow@met.fu-berlin.de'
    __short_description__ = "Creates a simple 2D countour map."
    __version__ = (0,0,1)
    #plugi.metadic is a special dictionary used for defining
       the plugin parameters
    __parameters__ =  parameters.ParameterDictionary(
                          parameters.File(name='file_path',
                             mandatory=True, help='The NetCDF
                              file to be plotted'),
                          parameters.Directory(name='
                             plot_name', default='
                             $USER_PLOTS_DIR/output', help='
                             The absolute paht to the
                             resulting plot'))

    def runTool(self, config_dict=None):
        file_path = config_dict['file_path']
        plot_name = config_dict['plot_name']

        result= self.call('module load ncl && ncl %s/plot.ncl
           file_path=\\"%s\\" plot_name=\\"%s\\"' % (self.
           getClassBaseDir(), file_path, plot_name))
        print result[0]
```

```
        #ncl adds the ".eps" suffix to our output file!
        plot_name += ".eps"
        if os.path.isfile(plot_name):
            return self.prepareOutput(plot_name)
```

That's certainly looks more daunting than it is. (If you haven't done the 5 Minutes Hands on tutorial do it now)
The difference with the dummy plug-in is that:

```
    In 1 we import the os package that will be used in 20 to
       check if the file was properly created.
    In 8 we provide a ParametersDictionary with some
       information about the parameters we expect.
    (Check the Reference section for more information about
       parameters and how it's used in the system)
    (You may refer directly to the source code: source:/src/
       evaluation_system/api/parameters.py)
    Here we basically define two input variables:
        file_path: a required file (just a string) with no
           default value.
        plot_name: An optional string for saving the resulting
           plot. By default it will be stored in a system
           managed directory with the name output.eps
    In 13 and 14 we read the variables passed to the
       application.
    In 16 we call the script (which should is assumed to be
       located in the same location as the script).
    (Check the code for more information source:/src/
       evaluation_system/api/plugin.py)
        self.call is a method from the inherited class that
           issues a shell command and return it's output as a
           tuple.
        self.getClassBaseDir() returns the location of the
           script.
        %s is used to format strings in Python, so "%s - %s" %
           (1, 2) == "1 - 2"
        \\" that's a double scape sequence... \\ tells python
           not to interpret the \ as enything special. It calls
            then bash with \" and we already saw why.
    in 17 we display the output from the call. By default
       stderr (All rerrors) are redirected to stdout so
       everything appears in the first element of the returned
       tuple (hence the result[0]
```

```
    in 19 we extend the named of the output file as ncl
       automatically adds that suffix.
    20 checks if the file exists and in such a case it will be
       returned using a special construct which checks the
       status of it and store some info for later use.
```

Running the plugin

We are done, now we have to test it. We will create a directory, store all files there (plot.ncl and nclplot.py) and tell the evaluation system to use the plugin.

```
$ mkdir plugin
$ cp nclplot.py plot.ncl plugin
$ cd plugin
$ ls
nclplot.py  nclplot.pyc plot.ncl
$ pwd
/home/user/plugin
$ export EVALUATION_SYSTEM_PLUGINS=/tmp/plugin,nclplot
```

Recalling from [5 Minutes Hands on]] the EVALUATION_SYSTEM_PLUGINS show the evaluation system where to find the code. It's a colon separated list of comma separated path, packages pairs. Basically:

```
EVALUATION_SYSTEM_PLUGINS=/some/path,mypackage:/oth/path,some.
    more.complex.package
```

We are done!

NCLPlot Usage

Now let's see what the evaluation system already can do with our simple plugin.
Let's start for displaying some help:

```
$ freva --plugin
NCLPlot: Creates a simple 2D countour map.
PCA: Principal Component Analysis
$ freva --plugin nclplot --help
NCLPlot (v0.0.1): Creates a simple 2D countour map.
Options:
file_path (default: None) [mandatory]
          The NetCDF file to be plotted

plot_name (default: $USER_PLOTS_DIR/output)
          The absolute paht to the resulting plot

$  freva --plugin nclplot
ERROR:  Missing required configuration for: file_path
```

So there you see to what purpose the metadict is being used. Let's use the $file and create a plot.

```
$ freva --plugin nclplot file_path=$file plot_name=first_plot
 Copyright (C) 1995-2012 - All Rights Reserved
 University Corporation for Atmospheric Research
 NCAR Command Language Version 6.1.0
 The use of this software is governed by a License Agreement.
 See http://www.ncl.ucar.edu/ for more details.

$ evince first_plot.eps
```

And evince should display the plot.

The configuration here is pretty simple, but you might still want to play around with it:

```
$ freva --plugin nclplot --save-config --config-file myconfig.
    cfg --dry-run file_path=$file plot_name=first_plot
INFO:__main__:Configuration file saved in myconfig.cfg
$ cat myconfig.cfg
[NCLPlot]
#: The absolute path to the resulting plot
plot_name=first_plot

#: [mandatory] The NetCDF file to be plotted
file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101-200012.
    nc

$ rm first_plot.eps
$ freva --plugin nclplot --config-file myconfig.cfg
$ evince first_plot.eps
```

As we say, it makes no sense here, but if you have multiple parameters you might want to let some predefined to suit your requirements. See also how to use the Evaluation System User Configuration.

Now the system also offers a history. There's a lot to be said about it, so I'll just leave a sample session here until I write a tutorial about it.

```
$ freva --history
21) nclplot [2013-01-11 14:44:15.297102] first_plot.eps {u'
    plot_name': u'first_plot', u'file_path': u'tas_Am...
22) ...
...
$ freva --history limit=1 full_text
```

```
21) nclplot v0.0.1 [2013-01-11 14:44:15.297102]
Configuration:
      file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101
         -200012.nc
      plot_name=first_plot
Output:
  /scratch/users/test/ncl_plot/plugin/first_plot.eps (available
     )

$ rm /scratch/users/test/ncl_plot/plugin/first_plot.eps
$ freva --history limit=1 full_text21) nclplot v0.0.1
   [2013-01-11 14:44:15.297102]
Configuration:
      file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101
         -200012.nc
      plot_name=first_plot
Output:
  /scratch/users/test/ncl_plot/plugin/first_plot.eps (deleted)
$ freva --history entry_ids=21 store_file=myconfig.cfg
Configuration stored in myconfig.cfg
$ cat myconfig.cfg [NCLPlot]
#: The absolute paht to the resulting plot
plot_name=first_plot

#: [mandatory] The NetCDF file to be plotted
file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101-200012.
   nc

$ freva --plugin NCLPlot --config-file myconfig.cfg
 Copyright (C) 1995-2012 - All Rights Reserved
 University Corporation for Atmospheric Research
 NCAR Command Language Version 6.1.0
 The use of this software is governed by a License Agreement.
 See http://www.ncl.ucar.edu/ for more details.

$ freva --history limit=2 full_text
22) nclplot v0.0.1 [2013-01-11 14:51:40.910996]
Configuration:
      file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101
         -200012.nc
      plot_name=first_plot
Output:
```

```
  /scratch/users/test/ncl_plot/plugin/first_plot.eps (available
     )
21) nclplot v0.0.1 [2013-01-11 14:44:15.297102]
Configuration:
      file_path=tas_Amon_MPI-ESM-LR_decadal1990_r1i1p1_199101
         -200012.nc
      plot_name=first_plot
Output:
  /scratch/users/test/ncl_plot/plugin/first_plot.eps (modified)

$ freva --history --help
...
```

Well, this is it. I'll be adding more tutorials doing some more advanced stuff but I think now you have the knowledge to start creating plugins for the whole community.

1.3 Parameters

Parameters are central to the plugin and they are managed at the evaluation_system.api.parameters module. Here a brief summary about what they can do and how they can be used.

ParameterDictionary

This class manages everything related to the set of parameters the plugin uses. It's an ordered dictionary so the order in which parameters are defined is the order in which they will be presented to the user in the help, configuration files, web forms, etc.

ParameterType

This is the central Class handling parameters. Normally you use another class that inherits functionality from this one, but as most of the functionality is define by this class, we will describe the options used in the constructor and therefore in all other classes inheriting from this one.

```
Option   Default Value   Description
name     None    Name of the parameter
default          None    The default value if none is provided
                         (this value will also be validated and
                            parsed, so it must be a
                         valid parameter value!)

mandatory        False   If the parameter is required (note that
    if there's a default value,
                         the user might not be required to set
                            it, and can always change it,
                         though he/she is not allowed to unset
                            it)
```

```
max_items       1       If set to > 1 it will cause the values
   to be returned in a list
                        (even if the user only provided 1). An
                           error will be risen if more
                        values than those are passed to the
                           plugin

item_separator   ,      The string used to separate multiple
   values for this parameter.
                        In some cases (at the shell, web
                           interface, etc) the user have always
                        the option to provide multiple values
                           by re-using the same parameter
                        name (e.g. param1=a param1=b produces
                           {'param1': ['a', 'b']}). But the
                        configuration file does not allow this
                           at this time. Therefore is better
                        to setup a separator, even though the
                           user might not use it while giving
                        input. It must not be a character, it
                           can be any string (make sure it's
                        not a valid value!!)

regex   .*              A regular expression defining valid "
   string" values before parsing them to
                        their defining classes (e.g. an Integer
                            might define a regex of "[0-9]+" to
                        prevent getting negative numbers). This
                            will be used also on Javascript
                        so don't use fancy expressions or make
                           sure they are understood by both
                        python and Javascript.

help    No help         The help string describing what this
   parameter is good for.

print_format    %s      A python string format that will be
   used when displaying the value of
                        this parameter (e.g. %.2f to display
                           always 2 decimals for floats)
```

```
Available Parameters
String:
```

```
    validated as string

Float

    validated as float

Integer

    validated as integer

File

    validated as string
    shows select file widget on the website

Directory

    validated as string
    system does autmatically append a unique id

InputDirectory

    validated as string
    system does not autmatically append a unique id

CacheDirectory

    validated as string

Bool

    validated as boolean
    shows radio buttons on the website

SolrField

    shows solr_widget on the website
    takes additional parameters:
        facet: CMOR facet (required)
        group: If you have more than one solr group in your
           plugin
                like MurCSS (default=1)
```

```
        multiple: Allow multiple selections (default=False)
        predefined_facets: Fix some facets for the data search
           (default=None)

SelectField

    only speficied values can be selected
    takes additional parameters:
        options: python dictionary with "key":"value" pairs
```

1.4 Special Variables

The plugin has access to some special parameters which are setup for the user running the plugin. These are:

```
Name    Description
$USER_BASE_DIR          Absolute path to the central directory
   for this user in
                        the evaluation system.
$USER_OUTPUT_DIR        Absolute path to where the output data
   for this user is stored.
$USER_PLOTS_DIR         Absolute path to where the plots for
   this user is stored.
$USER_CACHE_DIR         Absolute path to where the cached data
   for this user is stored.
$SYSTEM_DATE    Current date in the form YYYYMMDD (e.g.
   20120130).
$SYSTEM_DATETIME        Current date in the form
   YYYYMMDD_HHmmSS (e.g. 20120130_101123).
$SYSTEM_TIMESTAMP       Milliseconds since epoch (i.e. a new
   number every millisecond,
                        e.g. 1358929581838).
$SYSTEM_RANDOM_UUID     A random UUID string (e.g. 912cca21
   -6364-4f46-9b03-4263410c9899).
```

These can be used in the plugin like this:

```
#[...]
    __config_metadict__ =  metadict(compact_creation = True,
                                    output_file = '
                                       $USER_OUTPUT_DIR/${
                                       input_file}',
                                    output_plot = '
                                       $USER_PLOTS_DIR/${
                                       input_file}',
```

```
                                        work          = '
                                            $USER_CACHE_DIR',
                                        input_file    = (None, dict(
                                            mandatory=True, type=str
                                            ,
                                        help='The processed file'))
                                    )
```