Smart Script Class Library

The Smart Script Class has many library methods that are available to Smart Tools and Procedures. Complete documentation is within the class itself (viewable from the Edit Action Dialog Utilities Window) and has been duplicated here.

General Arguments

The following arguments are used throughout the SmartScript Library methods.

self: When you call a method, use the "self" prefix (see examples below)

model: There are various ways to specify the database model from which you want the values:
-- Simply "Fcst" or "Official" OR
-- siteID_type_model_modeltime
where the "type" is an empty string for Standard GFE data
and is "D2D" for D2D data.
Examples:
BOU__NAM_Mar2912 :gets March 29 12Z NAM run created by GFE.
BOU_D2D_ETA_Mar2912 :gets March 29 12Z original NAM run from D2D.
If you omit the "modeltime", the most recent model run will
be selected. For example:
BOU__NAM : gets the most recent NAM run created by GFE.
BOU_D2D_ETA : gets the most recent original ETA run from D2D.
-- the result of soliciting a model from the user using the "model" or "D2D_model" type of VariableList entry. (See examples above.)
-- you may also use a DatabaseID (see getDatabase, below)
-- simple string with no special characters (this will be assumed to be a model created "on-the-fly"

element: The element name in quotes: e.g. "QPF", "rh", "tp"

level: The level in quotes: e.g. "SFC", "MB350", "BL030"

x, y: integer coordinates

timeRange: Must be a special time range object such as that passed in the argument list as GridTimeRange

mode: specifies how to handle the situation if multiple grids are found within the given time range:
"TimeWtAverage": return time-weighted Average value
"Average" : return Average value
"Max" : return Max value
"Min" : return Min value
"Sum" : return Summed value
"First" : return value from grid with earliest time range
"List" : return list of values

noDataError: If 1, and there is no data, the Smart Tool will abort. Otherwise, return None. None is a special variable in Python which can be tested as follows:

```
PoP = self.getValue("Fcst", "PoP", "SFC", x,y, GridTimeRange, noDataError=0)
if PoP == None:
print "No data found for PoP"
```

mostRecentModel: Applies only to model data. Will get the most recent model and ignore any times (if included) in the model argument. (Note that if a time is not included in the model argument, you will automatically get the most recent model no matter how this argument is set.)

Grid Access Methods

getGrids

```
def getGrids(self, model, element, level, timeRange, mode="TimeWtAverage", noDataError=1, mostRecentModel=0):
    # Get the value(s) for the given model, element, and level
    # at the x, y coordinate and over the given timeRange.
    #
    # The resulting grid values can be accessed as follows:
    # PoPGrid = self.getGrids("Fcst","PoP","SFC", GridTimeRange)
    # popValue = PoPGrid[x][y]
    #
    # where x and y are integer grid coordinates.
```

taperGrid

```
def taperGrid(self, editArea, taperFactor=5):
    # Returns a 2-D Grid of values between 0-1 about the
    # given edit area.
    # These values can be applied by smart tools to taper results.
    #
    # Argument:
    # editArea : must be of type AFPS.ReferenceData or None
    # (use editArea tool argument)
    # taperFactor: If set to zero, will do Full Taper
    #
    # Example:
    # def preProcessTool(self, editArea):
    #     taperGrid = self.getTaperGrid(editArea, 5)
    # def execute(self, x, y):
    #     result = value * taperGrid[x][y]
    #
```

directionTaperGrid

```
def directionTaperGrid(self, editArea, direction):
    # Returns a 2-D Grid of values between 0-1 within the
    # given edit area.
    # E.g. if the Dir is W and x,y is half-way along the
    # W to E vector within the given edit area, the value of
    # directionTaperGrid at x,y will be .5
    # These values can be applied by smart tools to show
    # spatial progress across an edit area.
    #
    # Argument:
    # editArea : must be of type AFPS.ReferenceData or None
    # (use editArea tool argument)
    # direction : 16 point text direction e.g. "NNW", "NW", etc.
    #
```

```
    # Example:
    # def preProcessTool(self, editArea):
    #     spaceProgress = self.directionTaperGrid(editArea, "NW")
    # def execute(self, x, y):
    #     result = value * spaceProgress[x][y]
    #
```

getValue (point-based only)

```
def getValue(self, model, element, level, x, y, timeRange, mode="TimeWtAverage", noDataError=1,
mostRecentModel=0):
    # Point-based tools only.
    # Get the value(s) for the given model, element, and level
    # at the x, y coordinate and over the given timeRange.
    # Examples:
    # Get the following arguments in your execute method:
    # def execute(self, x, y, GridTimeRange, varDict):
    #
    # Accessing values from the Fcst database:
    #
    # T = self.getValue("Fcst", "T", "SFC", x, y, GridTimeRange)
    # PoP = self.getValue("Fcst", "PoP", "SFC", x,y, GridTimeRange,
    # noDataError=0)
    #
    # From an IFP model database:
    #
    # T = self.getValue("BOU__NAM_Oct3100", "T", "SFC",x,y,GridTimeRange)
    # T = self.getValue("BOU__NAM", "T", "SFC",x,y,GridTimeRange)
    #
    # Or, using a variable:
    #
    # VariableList = [("Model:" , "", "model")]
    # def execute(self, x, y, GridTimeRange, varDict):
    # model = varDict["Model:"]
    # T = self.getValue(model, "T", "SFC",x,y,GridTimeRange)
    #
    # From D2D database:
    #
    # rh = self.getValue(
    # "BOU_D2D_ETA", "rh", "MB850", x, y, GridTimeRange)
    # rh = self.getValue(
    # "BOU_D2D_ETA_Oct3100", "rh", "MB850",x, y, GridTimeRange)
    #
    # Or, using a variable:
    #
    # VariableList = [("Model:" , "", "D2D_model")]
    # def execute(self, x, y, GridTimeRange, varDict):
    # model = varDict["Model:"]
    # rh = self.getValue(model, "rh", "MB850", x, y, GridTimeRange)
    #
```

getComposite (numeric only)

```
def getComposite(self, WEname, GridTimeRange, exactMatch=1):
    # Returns a composite grid consisting of the primary grid and any
    # corresponding ISC grid, blended together based on the mask information
    # derived from the Grid Data History. Primary grid must exist. Returns
    # the set of points that are valid in the output grid. (Note the output
    # grid consists of the primary grid and isc grid. Any "invalid" points,
    # indicate those areas that have no isc data and are outside the home
    # site's region. The returned grid will have the primary data in
    # the site's region.)
    #
    # A Python tuple is returned.
    # For Scalar elements, the tuple contains:
    # a numeric grid of 1's and 0's where 1 indicates a valid point
    # a numeric grid of scalar values
    # For Vector elements, the tuple contains:
    # a numeric grid of 1's and 0's where 1 indicates a valid point
    # a numeric grid of scalar values representing magnitude
    # a numeric grid of scalar values representing direction
    # For Weather elements, the tuple contains:
    # a numeric grid of 1's and 0's where 1 indicates a valid point
    # a numeric grid of byte values representing the weather value
    # list of keys corresponding to the weather values
    #
    # For example:
    #    isc = self.getComposite(WEname, GridTimeRange)
    #    if isc is None:
    #        self.noData()
    #    # See if we are working with a Scalar or Vector element
    #    wxType = variableElement_GridInfo.type()
    #    if wxType == 0: # SCALAR
    #        bits, values = isc
    #    elif wxType == 1: # VECTOR
    #        bits, mag, dir = isc
```

getGridInfo

```
def getGridInfo(self, model, element, level, timeRange, mostRecentModel=0):
    #  Return a list of GridInfo objects for the given weather element and timeRange
    #   Example:
    #        timeRange = self.getTimeRange("Today")
    #        infoList = self.getGridInfo("Fcst", "T", "SFC", timeRange)
    #        for info in infoList:
    #                print "grid", info.gridTime()
```

Sounding Methods

makeNumericSounding (numeric only)

```
def makeNumericSounding(self, model, element, levels, timeRange, noDataError=1, mostRecentModel=0):
    # Make a numeric sounding for the given model, element, and levels
    # Example:
    # levels = ["MB850","MB800","MB750","MB700","MB650","MB600"]
    # gh_Cube, rh_Cube = self.makeNumericSounding(
    # model, "rh", levels, GridTimeRange)
    #
    # Arguments:
    #
    # The "levels" argument is a Python list of levels INCREASING
    # in height.
    # This method returns two numeric cubes:
    # ghCube of geopotential heights for the given levels
    # valueCube of values for the given levels
```

interpolateValues

```
def interpolateValues(self, height, (h1, v1), (h2, v2)):
    # Interpolate between the height and values
```

linear

```
def linear(self, xmin, xmax, ymin, ymax, we):
```

extrapolate

```
def extrapolate(self, height, (h1, v1),(h2, v2)):
    # Extrapolate from the height and values
```

interpolateScalarValues

```
def interpolateScalarValues(self, height, (h1, v1), (h2, v2)):
    # Interpolate between the height and values
```

interpolateVectorValues

```
def interpolateVectorValues(self, height, (h1, v1), (h2, v2)):
    # Interpolate between the height and values
```

getLevels

```
def getLevels(self, level1, level2, noDataError=1):
    # Return a list of levels between and including level1 and level2
    # Will do ascending or descending depending on order of arguments
    # levels = self.getLevels("MB900", "MB500") # descending
    # levels = self.getLevels("MB600", "MB1000") # ascending
```

getSoundingValue (point-based only)

```
def getSoundingValue(self, model, element, levels, x, y, timeRange, height, method=None, noDataError=1,
mostRecentModel=0):
    # Make a sounding for the given model, element, and levels at
    # the x,y coordinates for the given timeRange.
    # Then get the value of the sounding for the given height.
    # Use the given method for calculating a value between two heights
    # in the sounding.
```

```
    # The levels MUST be increasing in height.
    # Height needs to be in the same units as the sounding.
    # The default method if None is specified, is "interpolateValues"
    #
    # Example:
    # topo_M = convertFtToM(Topo)
    # levels = ["MB850","MB800","MB750","MB700","MB650","MB600"]
    # surfaceRH = self.getSoundingValue(model, "rh", levels, x, y,
    # GridTimeRange, topo_M)
    # Arguments:
    #
    # The "levels" argument is a Python list of levels INCREASING
    # in height.
    # The other argument descriptions are the same as for
    # "getValue" (see above)
    # Find bounding levels (lowerLevel and upperLevel)
    # for the given height
```

makeSounding (point-based only)

```
def makeSounding(self, model, element, levels, x, y, timeRange, noDataError=1, mostRecentModel=0):
    # Make a sounding for the given model, element, and levels
    # Example:
    # levels = ["MB850","MB800","MB750","MB700","MB650","MB600"]
    # sounding = self.makeSounding(model, "rh", levels, x, y,
    # GridTimeRange)
    #
    # Arguments:
    #
    # The "levels" argument is a Python list of levels INCREASING
    # in height.
    # The other argument descriptions are the same as for
    # "getValue" (see above)
    # This method returns a tuple of height, value tuples.
    # In general, you will not need to access the sounding directly,
    # but will simply pass it to the "getValueFromSounding" method
    # (see below)
```

getValueFromSounding (point-based only)

```
def getValueFromSounding(self, sounding, height, method=None):
    # Get the value of the sounding for the given
    # height. Use the given method for calculating a value
    # between two heights in the sounding.
    # The default method if None is specified, is "interpolateValues"
    #
    # Example:
    # levels = ["MB850","MB800","MB750","MB700","MB650","MB600"]
    # sounding = self.makeSounding(model, "rh", levels, x, y,
    # GridTimeRange)
    # topo_M = convertFtToM(Topo)
    # surfaceRH = self.getValueFromSounding(sounding, topo_M)
```

```
    #
    #
    # Height needs to be in the same units as the sounding
    # The sounding is assumed to be increasing in height
```

getNumericMeanValue (numeric only)

```
def getNumericMeanValue(self, model, element, levels, timeRange, noDataError=1):
    # Return a numeric array of mean values for the given element between and including
    # the given levels
```

getMeanValue (point-based only)

```
def getMeanValue(self, model, element, levels, x, y, timeRange, noDataError=1):
    # Get a mean value for the given element between and including
    # the given levels
```

Conversion Methods

UVToMagDir

```
def UVToMagDir(self, u, v):
```

MagDirToUV

```
def MagDirToUV(self, mag, dir):
```

convertMsecToKts

```
def convertMsecToKts(self, value_Msec):
    # Convert from meters/sec to Kts
    # meters = 1.0/.0006214 * miles
    # seconds = 60*60*hours
    # m/s = (1.0/.0006214) * (1/ 60*60) * mph
```

convertKtoF

```
def convertKtoF(self, t_K):
    # Convert the temperature from Kelvin to Fahrenheit
    # Degrees Fahrenheit = (Degrees Kelvin - 273.15) / (5/9) + 32
```

KtoF

```
def KtoF(self, t_K):
```

convertFtoK

```
def convertFtoK(self, t_F):
    # Convert the temperature from Kelvin to Fahrenheit
    # Degrees Kelvin = (Degrees Fahrenheit - 32) * (5 / 9) + 273.15
```

FtoK

```
def FtoK(self, t_F):
```

convertFtToM

```
def convertFtToM(self, value_Ft):
    # Convert the value in Feet to Meters
```

round

```
def round(self, value, increment, mode="Nearest"): "Round according to mode: Up, Down, Nearest and increment"
```

Error Handling

abort

```
def abort(self, info):
    # This call will send the info to the GFE status bar,
    # put up a dialog with the given info, and abort the
    # smart tool or procedure.
    # Example:
    # self.abort("Error processing my tool")
    #
```

noData

```
def noData(self, info="Insufficient Data to run Tool"):
    #Raise the NoData exception error
```

cancel

```
def cancel(self):
    # Cancels a smart tool without displaying an error message
```

errorReturn

```
def errorReturn(self, noDataError, message):
```

statusBarMsg

```
def statusBarMsg(self, message, status):
    # Sends the text message to the GFE status bar with the
    # given status code: "R" (regular), "S" (significant), or "U" (urgent)
    # Example:
    # self.statusBarMsg("Running Smart Tool", "R")
    #
```

Procedure Methods

Command Arguments

```
    # These commands always apply to the mutable model only.
    # name1, name2, name3 is a list of the weather element names
    # startHour is the starting hour for the command offset from modelbase
    # endHour is the ending hour for the command offset from modelbase.
    # The ending hour is NOT included in the processing of the
    # command.
    # modelbase is the name of the model to be used to determine base times
    # Note that if this is "", then 0000z from today will be
    # used for the base time.
    # modelsource is the name of the model to be used in the copy command
    # copyOnly is 0 for move and 1 for copy only in the time shift command
    # hoursToShift is the number of hours to shift the data in time
    # shift command
    # databaseID must be of type AFPS.DatabaseID
    # Can be obtained in various ways:
    # --By calling findDatabase (see below)
    # --By calling getDatabase (see below) with the result
    # of a VariableList entry of type "model" or "D2D_model"
```

```
    # timeRange must be of type AFPS.TimeRange.
    # Can be obtained in various ways:
    # --As an argument passed into Smart Tool or Procedure,
    # --By calling getTimeRange (see below)
    # --By calling createTimeRange (see below)
```

copyCmd

```
def copyCmd(self, elements, databaseID, timeRange):
    # copyCmd(['name1', 'name2', 'name3'], databaseID, timeRange)
    # Copies all grids for each weather element from the given database
    # into the weather element in the mutable database that overlaps
    # the time range.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #     self.copyCmd(['T', 'Wind'], databaseID, timeRange)
    # will copy the Temperature and Wind fields analysis through 48 hours
    # from the latest NAM and place them into the forecast.
    #
```

copyToCmd

```
def copyToCmd(self, elements, databaseID, timeRange):
    # copyCmd([('srcName1', 'dstName1), ('srcName2', 'dstName2')], databaseID, timeRange)
    # Copies all grids for each weather element from the given database
    # into the weather element in the mutable database that overlaps
    # the time range. Both the source
    # and destination names are given.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #     self.copyToCmd([('T', 'MaxT'), ('MaxT', 'MinT')], databaseID, timeRange)
    # will copy T to MaxT and MaxT to MinT
    # from the latest NAM and place them into the forecast.
    #
```

deleteCmd

```
def deleteCmd(self, elements, timeRange):
    # deleteCmd(['name1', 'name2', 'name3'], timeRange)
    # Deletes all grids that overlap the input time range for element
    # in the mutable database.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #      self.deleteCmd(['T', 'Wind'], databaseID, timeRange)
    # will delete the Temperature and Wind fields analysis up to
    # but not including 48 hours relative to the start time of
    # the latest NAM model.
    #
```

zeroCmd

```
def zeroCmd(self, elements, timeRange):
    # zeroCmd(['name1', 'name2', 'name3'], timeRange)
    # Assigns the minimum possible value for scalar and vector, and ""
    # for weather for the parameter in the mutable database for all grids
    # that overlap the specified time range.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #     self.zeroCmd(['T', 'Wind'], databaseID, timeRange)
    # will zero the Temperature and Wind grids through 48 hours
    # relative to the start time of the latest NAM model.
    #
```

interpolateCmd

```
def interpolateCmd(self, elements, timeRange, interpMode="GAPS", interpState="SYNC", interval=0, duration=0):
    # interpolateCmd(['name1', 'name2', 'name3'], timeRange,
    # interpMode="GAPS", interpState="SYNC", interval=0, duration=0)
    # Interpolates data in the forecast for the named weather elements
    # for the given timeRange.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #     self.interpolateCmd(['T', 'Wind'], timeRange, "GAPS","SYNC")
    # will interpolate the Temperature and Wind grids up to but
    # but not including 48 hours relative to the start time of
    #the latest NAM model.
    # The interpolation will run in SYNC mode i.e. completing before
    # continuing with the procedure.
    #
```

createFromScratchCmd

```
def createFromScratchCmd(self, elements, timeRange, repeat=0, duration=0):
    # createFromScratchCmd(['name1', 'name2'], timeRange, repeat, duration)
    # Creates one or more grids from scratch over the given timeRange
    # and assigns the default (minimum possible value for scalar
    # and vector, "" for weather).
    # The repeat interval and duration (both specified in hours) are
    # used to control the number of grids created. If 0 is specified for
    # either one, than only 1 grid is created for the given time range. If
    # valid numbers for duration and repeat are given, then grids will
    # be created every "repeat" hours and they will have a duration
    # of "duration" hours. If there is not enough room remaining to create
    # a grid with the full duration, then no grid will be created in the space
    # remaining. If you don't get the desired results, be sure that your input
    # time range starts on a valid time constraint for the element. If the
    # element's time constraints (not the values supplied in this routine) contains
    # gaps (i.e., duration != repeatInterval), then the repeat interval and
    # duration will be ignored and grids will be created for each possible
    # constraint time.
    # Example:
```

```
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #     timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #     self.createFromScratchCmd(['T', 'Wind'], timeRange, 3, 1)
    # will create the 1-hour Temperature grids through 48 hours at
    # 3 hour intervals relative to the start time of the latest NAM model.
    #
```

timeShiftCmd

```
def timeShiftCmd(self, elements, copyOnly, shiftAmount, timeRange):
    # timeShiftCmd(['name1', 'name2'], copyOnly, shiftAmount, timeRange)
    # Performs a time shift by the shiftAmount for all elements that
    # overlap the time range.
    # Example:
    #     databaseID = self.findDatabase("NAM") # Most recent NAM model
    #      timeRange = self.createTimeRange(0, 49, "Database", databaseID)
    #      self.timeShiftCmd(['T', 'Wind'], 1, 3, timeRange)
    #
```

splitCmd

```
def splitCmd(self, elements, timeRange):
    # splitCmd(elements, timeRange)
    # Splits any grid that falls on the start time or ending time of the
    # specified time range for the given parameter in the mutable database.
    #
```

fragmentCmd

```
def fragmentCmd(self, elements, timeRange):
    # fragmentCmd(elements, timeRange)
    # Fragments any grids that overlap the input time range for the parm
    # identified in the mutable database.
    #
```

assignValueCmd

```
def assignValueCmd(self, elements, timeRange, value):
    # assignValueCmd(elements, timeRange, value)
    # Assigns the specified value to all grids points for the grids that
    # overlap the specified time range, for the weather element in the mutable
    # database specified.
    # value is:
    # an Integer or Float for SCALAR
    # a magnitude-direction tuple for VECTOR: e.g. (55,120)
    # a text string for Weather which can be obtained via the
    # WxMethods WxString method
    # Example:
    #
    # Scalar
    #     value = 60
    #     self.assignValue(["T","Td"], 0, 12, 'NAM', value)
    #
    # Vector
```

```
    #     value = (15, 120)
    #     self.assignValue(["Wind"], 0, 12, 'NAM', value)
    #
    # Weather
    #     from WxMethods import *
    #      value = WxString("Sct RW")
    #      self.assignValue(["Wx"], 0, 12, 'NAM', value)
```

Calling Smart Tools and Procedures Arguments

```
    ## editArea : must be of type AFPS.ReferenceData or None
    ##     (See getEditArea)
    ##      If you specify None, the system will supply
    ##      the active edit area from the GFE or from
    ##      the editArea argument for runProcedure.
    ## timeRange: must be of type AFPS.TimeRange or None
    ##      (See getTimeRange and createTimeRange)
    ##      If you specify None, the system will supply
    ##      the selected Time Range from the GFE or from
    ##      the timeRange argument for runProcedure.
    ## varDict : If you supply a varDict in this call, the
    ##      variable list dialog will not be displayed
    ##      when the tool is run.
    ##      If you supply a varDict from a Procedure,
    ##      make sure that the variables
    ##      for all the tools called by the procedure are
    ##      supplied in your varDict.
    ## missingDataMode: Can be "Stop", "Skip", or "Create". If not
    ##      included, will be set to the current GFE default.
    ## modal: If 0, VariableList dialogs will appear with the
    ##      non-modal "Run" and "Run/Dismiss" buttons.
    ##      Otherwise, they will appear with the "Ok" button.
    ##
    ## These commands all return an error which will be None if no
    ##      errors occurred. Otherwise, the errorType and errorInfo
    ##      can be accessed e.g. error.errorType() and error.errorInfo()
    ## If "noData" has been called, the errorType will be "NoData" and
    ##      can be tested by the calling tool or script.
```

callSmartTool

```
def callSmartTool(self, toolName, elementName, editArea=None, timeRange=None, varDict=None, editValues=1,
calcArea=0, calcGrid=0, passErrors=[], missingDataMode=AFPS.EditActionMsg.MD_DEFAULT, modal=1):
     # passErrors: a list of errors to ignore and pass back to the
     # calling program. Some errors that can be ignored are:
     #     NoData
     #     NoElementToEdit
     #     ExecuteOrClassError
     #     LockedGridError
     #
     # For example:
     # In the Procedure:
```

```
#     error = self.callSmartTool(
#         "MyTool", "MixHgt", editArea, timeRange, varDict,
#         passErrors= ["NoData"])
#     if error is not None:
#         print "No Data available to run tool"
#
# In the Smart Tool:
#     mixHgt = self.getGrids(model, "MixHgt", "SFC", timeRange)
#     if mixHgt is None:
#          self.noData()
```

callProcedure

```
def callProcedure(self, name, editArea=None, timeRange=None, varDict=None,
missingDataMode=AFPS.EditActionMsg.MD_DEFAULT, modal=1):
```

Creating On-the-Fly Grids

createGrid

```
    def createGrid(self, model, element, elementType, numericGrid, timeRange,
                   descriptiveName=None, timeConstraints=None,
                   precision=None, minAllowedValue=None,
                   maxAllowedValue=None, units=None, rateParm=0,
                   discreteKeys=None, discreteOverlap=None):


        # Creates a grid for the given model and element.
        # If the model and element do not already exist, creates them on-the-fly       #
        # The descriptiveName, timeConstraints, precision, minAllowedValue,
        # maxAllowedValue, units, rateParm, discreteKeys, discreteOverlap,
        # only need to be specified for the first grid being created.  These
        # values are ignored for subsequent calls to createGrid() for
        # the same weather element.

        # DISCRETE elements require a definition for discreteKeys and
        # discreteOverlap. For DISCRETE, the precision, minAllowedValue,
        # maxAllowedValue, and rateParm are ignored.

        # Note that this works for numeric grids only.
        # The arguments exampleModel, exampleElement, and exampleLevel can be
        # supplied so that the new element will have the same characteristics
        # (units, precision, etc.) as the example element.
        #
        # model -- If you are creating an "on-the-fly" element (i.e. not
        #          in the server), this should be a simple string with
        #          with no special characters.  The site ID and other
        #          information will be added for you.
        #          If you are creating a grid for a model that exists
        #          in the server, follow the guidelines for the model
        #          argument described for the "getValue" command.
        # element -- This should be a simple string with no special
        #          characters.
```

```
    # elementType -- "SCALAR", "VECTOR", "WEATHER", or "DISCRETE"
    # numericGrid -- a Numeric Python grid
    # timeRange -- valid time range for the grid.  You may want
    #         to use the "createTimeRange" command
    #
    # The descriptiveName, timeConstraints, precision, minAllowedValue,
    # precision, minAllowedValue, maxAllowedValue, and units can be
    # used to define the GridParmInfo needed. Note that timeConstraints
    # is not the C++ version, but a (startSec, repeatSec, durSec).
    #
    # Example:
    #    self.createGrid("ISCDisc", WEname+"Disc", "SCALAR", maxDisc,
    #                    GridTimeRange, descriptiveName=WEname+"Disc")
    #
```

deleteGrid

```
def deleteGrid(self, model, element, level, timeRange):
    # Deletes any grids for the given model and element
    # completely contained in the given timeRange.
    # If the model and element do not exist or if there are no existing grids,
    # no action is taken.
```

highlightGrids

```
def highlightGrids(self, model, element, level, timeRange, color, on=1):
    # Highlight the grids in the given time range using designated
    # color. If "on" is 0, turn off the highlight.
```

makeHeadlineGrid

```
def makeHeadlineGrid(self, headlineTable, fcstGrid, headlineGrid = None):
    # This method defines a headline grid based on the specified data.
    # The headlineTable parameter must be a list of tuples each containing
    # the threshold for each headline category and headline label
    # Example:
    #    headlineTable =[(15.0, 'SmCrftHSADV'),
    #                    (21.0, 'SmCrftADV'),
    #                    (34.0, 'GaleWRN'),
    #                    (47.0, 'StormWRN'),
    #                    (67.0, 'HurcnFrcWindWRN'),
    #                    ]
    # "fsctGrid" is the grid that defines what headline category should
    # be assigned. "headlineGrid" is the grid you wish to combine with
    # the calculated grid.  This forces a combine even if the GFE is not
    # in combine mode.  Omitting "headlineGrid" will cause the calculated
    # grid to replace whatever is in the GFE, no matter what the GFE's
    # combine mode. Note that a side effect of omitting the headline grid
    # is that the GFE will end up in replace mode after the tool completes.
```

Utilities

findDatabase

```
def findDatabase(self, databaseName, version=0):
    # Return an AFPS.DatabaseID object.
    # databaseName can have the appended type. E.g. "NAM" or "D2D_NAM"
    # version is 0 (most recent), -1 (previous), -2, etc.
    # E.g.
    # databaseID = self.findDatabase("NAM",0)
    # returns most recent NAM model
```

getDatabase

```
def getDatabase(self, databaseString):
    # Return an AFPS.DatabaseID object.
    # databaseString is the result of a VariableList entry of type
    # "model" or "D2D_model"
```

getTimeRange

```
def getTimeRange(self, timeRangeName):
    # Returns an AFPS.TimeRange object given a time range name
    # as defined in the GFE
    # E.g.
    # timeRange = self.getTimeRange("Today")
```

createTimeRange

```
def createTimeRange(self, startHour, endHour, mode="LT", dbID=None):
    # Returns an AFPS.TimeRange object given by:
    # startHour, endHour
    # (range is startHour up to and not including endHour)
    # startHour and endHour are relative to midnight of the
    # current day either in Local or Zulu time (see below)
    # mode can be:
    # "LT" : the startHour and endHour are relative to local time
    # "Zulu": relative to Zulu time,
    # "Database": relative to a database (e.g. model time.
    # In this case, the databaseID for the model must
    # be supplied (see findDatabase)
    #
    # E.g.
    # timeRange = self.createTimeRange(0,121,"Zulu")
    # databaseID = self.findDatabase("NAM")
    # timeRange = self.createTimeRange(120,241,"Database",databaseID)
```

getSamplePoints

```
def getSamplePoints(self, sampleSetName=None):
    # Return a list of x,y tuples representing sample points
    # sampleSet is the name of a saved sample set
    # if sampleSet is None, the sample points will be
    # those currently displayed on the GFE
```

_timeRangeStr

```
def _timeRangeStr(self, tr):
    # Returns given time range in format: "%Y%m%d_%H%M"
```

fformat

```
def fformat(self, value, roundVal):
    # Return a string for the floating point value
    # truncated to the resolution given by roundVal
```

dayTime

```
def dayTime(self, timeRange, startHour=6, endHour=18):
    # Return 1 if start of timeRange is between the
    # startHour and endHour, Return 0 otherwise.
    # Assume timeRange is GMT and convert to local time.
```

determineTimeShift

```
def determineTimeShift(self):
    # Returns the difference: Local time - GMT in seconds
```

getEditArea

```
def getEditArea(self, editAreaName):
    # Returns an AFPS.ReferenceData object given an edit area name
    # as defined in the GFE
```

saveEditArea

```
def saveEditArea(self, editAreaName, refData):
    # Saves the AFPS.ReferenceData object with the given name
```

setActiveEditArea

```
def setActiveEditArea(self, area):
    # Set the AFPS.ReferenceData area to be the active one in the GFE
    # Note: This will not take effect until AFTER the smart tool or
    # procedure is finished executing.
```

clearActiveEditArea

```
def clearActiveEditArea(self):
    # Clear the active edit area in the GFE
```

setActiveElement

```
def setActiveElement(self, model, element, level, timeRange, colorTable=None, minMax=None, fitToData=0):
    # Set the given element to the active one in the GFE
    # A colorTable name may be given.
    # A min/max range for the colorTable may be given.
    # If fitToData = 1, the color table is fit to the data
    #
    # Example:
    # self.setActiveElement("ISCDisc", WEname+"Disc", "SFC", GridTimeRange,
    # colorTable="Discrepancy", minMax=(-20,+20),
    # fitToData=1)
    #
```

getActiveElement

```
def getActiveElement(self):
    # Return the parm for the current active element
```

getGridCellSwath

```
def getGridCellSwath(self, editArea, cells):
    # Returns an AFPS.ReferenceData swath of the given
    # number of cells around the given an edit area.
    # The edit area must not be a query.
```

getLatLon

```
def getLatLon(self, x, y):
    # Get the latitude/longitude values for the given grid point coords = AFPS.CC2Dint(x,y) cc2D =
self.__gridLoc.latLonCenter(coords) return cc2D.y, cc2D.x
```

getGridCell

```
def getGridCell(self, lat, lon):
    # Get the corresponding x,y values for the given lat/lon
    # Return None, None if the lat/lon is outside the grid domain
```

getGrid2DBit

```
def getGrid2DBit(self, editArea):
    # Returns a Grid of on/off values indicating whether
    # or not the grid point is in the given edit area.
    # This could be used as follows in a Smart Tool:
    # def preProcessGrid(self):
    # editArea = self.getEditArea("Area1")
    # self.__area1Bits = self.getGrid2DBit(editArea)
    # editArea = self.getEditArea("Area2")
    # self.__area2Bits = self.getGrid2DBit(editArea)
    #
    # def execute(self, x, y):
    # if self.__area1Bits.get(x,y) == 1:
    #
    # elif self.__area2Bits.get(x,y) == 1:
    #
    #
```

getGridTimes

```
def getGridTimes(self, model, element, level, startTime, hours):
    # Return the timeRange and gridTimes for the number of hours
    # FOLLOWING the given startTime
```

getExprName

```
def getExprName(self, model, element, level="SFC", mostRecent=0):
    # Return an expressionName for the element
    # This method is complicated because it is handling all the
    # variations for the "model" argument. For a description
    # of the variations, see the "getValue" documentation above.
```

getModelName

```
def getModelName(self, databaseString):
    # Return the model name.
    # databaseString is the result of a VariableList entry of type
    # "model" or "D2D_model"
```

getD2Dmodel

```
def getD2Dmodel(self, model):
    # Given a GFE Surface model, return the corresponding D2D model
```

getParm

```
def getParm(self, model, element, level, timeRange=None, mostRecent=0):
    # Returns the parm object for the given model, element, and level
```

cacheElements

```
def cacheElements(self, elementNames):
    # Set these elements to be cached self.__parmMgr.addCachedParmList(elementNames)
```

unCacheElements

```
def unCacheElements(self, elementNames):
    # Set these elements to be un-cached
```

loadWEGroup

```
def loadWEGroup(self, groupName):
    # Load the given weather element group into the GFE
```

saveElements

```
def saveElements(self, elementList):
    # Save the given Fcst elements to the server
    # Example:
    # self.saveElements(["T","Td"])
```

publishElements

```
def publishElements(self, elementList, timeRange):
    # Publish the given Fcst elements to the server
    # over the given time range
    # Example:
    # self.publishElements(["T","Td"], timeRange)
```

combineMode

```
def combineMode(self):
    # returns 1 if we are in COMBINE mode and 0 if not
```

setCombineMode

```
def setCombineMode(self):
    # call it like this
    self.setCombineMode("Replace")

     #or

     self.setCombineMode("Combine")
```

getVectorEditMode

```
def getVectorEditMode(self):
    # Returns Vector Edit mode in the GFE
    # mode:
    #    "Magnitude Only"
    #    "Vector Only"
    #    "Both"
```

setVectorEditMode

```
def setVectorEditMode(self, mode):
    # Sets the Vector Edit mode in the GFE
    # mode:
    #    "Magnitude only"
    #    "Vector only"
    #    "Both"
```

esat

```
def esat(self, temp):
```

Numeric Library Methods

getTopo

```
def getTopo(self):
   # Return the numeric topo grid
```

wxMask

```
def wxMask(self, wx, query, isreg=0):
    # Returns a numeric mask i.e. a grid of 0's and 1's
    # where the value is 1 if the given query succeeds
    # Arguments:
    #    wx -- a 2-tuple:
    #       wxValues : numerical grid of byte values
    #       keys : list of "ugly strings" where the index of
    #          the ugly string corresponds to the byte value in
    #          the wxValues grid.
    #    query -- a text string representing a query
    #    isreg -- if 1, the query is treated as a regular expression
    #       otherwise as a literal string
    # Examples:
    #    # Here we want to treat the query as a regular expression
    #    PoP = where(self.wxMask(wxTuple, "^Chc:", 1), maximum(40, PoP), PoP)
    #    # Here we want to treat the query as a literal
    #    PoP = where(self.wxMask(wxTuple, ":L:") maximum(5, PoP), PoP)
    #
```

discreteMask

```
def wxMask(self, wx, query, isreg=0):
    # Returns a numeric mask i.e. a grid of 0's and 1's
    # where the value is 1 if the given query succeeds
    # Arguments:
    #    wx -- a 2-tuple:
```

```
    #       values : numerical grid of byte values
    #       keys : list of "ugly strings" where the index of
    #          the ugly string corresponds to the byte value in
    #          the discreteValues grid.
    #    query -- a text string representing a query
    #    isreg -- if 1, the query is treated as a regular expression
    #       otherwise as a literal string
    # Examples:
    #    # Here we want to treat the query as a regular expression
    #    PoP = where(self.wxMask(wxTuple, "BlzrdWRN", 1), maximum(40, PoP), PoP)
    #    # Here we want to treat the query as a literal
    #    PoP = where(self.wxMask(wxTuple, "WintStmWRN") maximum(5, PoP), PoP)
    #
```

getIndex

```
def getIndex(self, uglyStr, keys):
    #    Returns the byte value that corresponds to the
    #    given ugly string. If the ugly string is not found a new key will be added
    #    and the index to the new entry will be returned.  This method works for grids
    #    of type weather and discrete.
    # Arguments:
    #    uglyStr: a string representing a weather value or a discrete value
    #    keys: a list of ugly strings.
    #       A Wx argument represents a 2-tuple:
    #       wxValues : numerical grid of byte values
    #       keys : list of "ugly strings" where the index of
    #        the ugly string corresponds to the byte value in the wxValues grid.
    #       For example, if our keys are:
    #          "Sct:RW:-::"
    #          "Chc:T:-::"
    #          "Chc:SW:-::"
    #       Then, the wxValues grid will have byte values of 0 where
    #        there is "Sct:RW:-::", 1 where there is "Chc:T:-::"
    #        and 2 where there is "Chc:SW:-::"
    #
```

encodeEditArea

```
def encodeEditArea(self, editArea):
    # Returns a Numeric Python mask for the edit area
    # "editArea" can be a named area or a referenceData object
```

decodeEditArea

```
def decodeEditArea(self, mask):
    # Returns a refData object for the given mask
```

getindicies

```
def getindicies(self, o, l):
```

offset

```
def offset(self, a, x, y):
    # Gives an offset grid for array, a, by x and y points
```

agradient

```
def agradient(self, a):
    # Gives offset grids in the "forward" x and "up" y directions
```

diff2

```
def diff2(self, x, n=1, axis=-1):
    # diff2(x,n=1,axis=-1) calculates the first-order, discrete
    # center difference approximation to the derivative along the axis
    # specified. array edges are padded with adjacent values.
```

Python Object Methods

These methods allow you to save, get, and delete Python objects to and from the server.

saveObject

```
def saveObject(self, name, object, category):
    # Save a Python object (e.g. a Numeric grid)
    # in the server under the given name
    # Example:
    # self.saveObject("MyGrid", numericGrid, "DiscrepancyValueGrids")
    #
```

getObject

```
def getObject(self, name, category):
    # Returns the given object stored in the server
    # Example:
    # discrepancyValueGrid = self.getObject("MyGrid","DiscrepancyValueGrids")
    #
```

deleteObject

```
def deleteObject(self, name, category):
    # Delete the given object stored in the server
    # Example:
    # self.deleteObject("MyGrid", "DiscrepancyValueGrids")
    #
```