ObitTalk Software Documentation

Draft version: 0.0 September 30, 2016

Abstract

This documents describes the ObitTalk interface to AIPS and Obit software. ObitTalk is a python package which allows running AIPS and Obit tasks and direct access to astronomical data and application libraries using Obit. ObitTalk supports both local, distributed and remote (over a network) data processing.

Contents

1.1 Introduction

ObitTalk is derived from the ParselTongue project at JIVE written by Mark Kettenis and provides a scripting and interactive command line interface to astronomical data and processing software. In particular, AIPS and FITS data structures as used in the AIPS and Obit software packages are supported as well as AIPS tasks and Obit tasks and class libraries and other python enabled software.

ObitTalk can start tasks and run scripts either locally or on a remote machine which has an ObitTalkServer process running. Some data access is supported through the AIPSUVData, AIPSImage, FITSUVData and FITSImage classes. Currently other interactive python functions only work locally.

Tasks, scripts and more detailed access to, and manipulation of, data are available. These are described below. Later sections give the interface details of the major functions.

1.2 Obtaining Software

Obit and related software is available from http://www.cv.nrao.edu/ bcotton/Obit.html. At present there is no system of stable releases so using the anonymous CVS interface is recommended. Obit depends heavily on third party software which is described on this page. The components of the Obit/ObitTalk package are:

- Obit
  Basic obit package and the support for radio interferometry
- ObitSD
  Obit "On The Fly" (OTF) single dish imaging package.
- ObitView
  Image display used by Obit.
- ObitTalk
  Scripting and interactive interface to Obit software.

These software packages come with installation instructions and config scripts to build them.

1.3 Architectural Design

ObitTalk is intended to allow local, remote and distributed processing of astronomical data with both interactive and scripting interfaces. ObitTalk is python with application specific modules loaded to allow access to astronomical data and software.

ObitTalk allows use of both external AIPS and Obit tasks as well as access to data structures and the high level Obit classes and functions. Tasks, scripts and to a lesser degree data access can be either performed locally or on a remote computer. Access to Obit class libraries is currently only available for data files visible on the computer on which the interactive/scripting python process is run. The various aspects of ObitTalk are discussed in the following.

1.3.1 Client–Server Organization

In the modern computing environment it is frequently desirable to have a user or scripting front end with data and a compute server at a remote site or another node of a cluster. ObitTalk implements

remote access by a client–server architecture for much (but not all) of its functionality. This is implemented by a split between the client and the server (internally called "Proxy") sides. If the client and server are on separate hosts, they are independent python processes communicating via xmlrpc. This protocol allows communication over networks using http protocols and can, in principle, be quite secure. If the client and server are on the same host, this is implemented in a single python process but with the same interface as in the remote server case.

The location of data is defined in terms of "disks", the definition of which includes a URL for data on remote server machines. ObitTalk thus knows where the data resides and does any operation and runs tasks needing it there. This mechanism also allows distributed processing over a LAN or cluster. The remote execution of scripts allows all functionality to be available on remote as well as local hosts.

1.3.2 Tasks

Tasks represent a relatively simple interface. They are external, independent executables which accept parameters and data files and may modify or produce other data files or parametric results. Currently only AIPS and Obit tasks are implemented but others which follow this model could be.

Obit is intended to be fully interoperable with AIPS and AIPS data structures are one possible "native" data type for Obit. AIPS and Obit share a common data and calibration model. This allows AIPS and Obit tasks to both be employed in the same processing session or script.

All access to tasks is through task objects which contain the relevant parameters and have methods to start, abort, wait for task execution, etc. Task execution is performed on the server (proxy) side. Starting tasks consists of spawning an external process and giving it the task parameters. These parameters are generally a combination of command line arguments and parameters in a file which the task process reads. Messages produced by the process on stdout or stderr are captured and passed back to the client. When a task is finished, it may write results back into a disk file which is then read by the waiting python proxy and returned to the client side. For local execution, the proxy is implemented in the same address space as the client python process. For remote execution, the proxy is a python server process on the remote machine.

1.3.3 Scripts

Scripts are implemented in the ObitScript class which is derived from the Task class and thus shares many properties. An ObitScript can execute a script which is a character string containing a sequence of python statements; these scripts have full access to the Obit/python bindings for data local to the execution host. An ObitScript can be executed either locally or remotely and either synchronously or asynchronously. ObitTalk itself can be given a script as a command line argument and this feature is used in the implementation of script execution. The script is wrapped in python code to properly initialize and shut down Obit and then written to a file. An external process is than started running ObitTalk on this script file.

1.3.4 Access to Obit class functions

Major Obit classes and functions have bindings to python and are thus available from python. All access to Obit c software is through a single dynamic library, Obit.so. Separate Obit related packages (Single Dish Obit) will have separate libraries but all Obit code must be liked into a single dynamic library. Since c and python have such very different view of data structures, the interface is complex. Python bindings to the major Obit classes and functions are implemented in the interface.

The python interface is defined in the Obit python subdirectory. The swig utility is used to interface the c library to python. The swig interface is defined in the ObitTypeMaps.swig (defines interface types) and *.inc which defines a python callable interface. The ObitTypeMaps.swig and *.inc files are concatenated by the Makefile before running swig. This generates a single, large shared library module to be imported into python. However, this is necessary to get all the classes into the same address space as Obit is dependent on class structures defining function pointers. A more elegant solution may be possible.

Note about swig. The maintainers of swig have modified the interface in more recent versions (after version 1.1?) so the output of swig, python/Obit_wrap.c, is distributed with other source code. The make procedure should not attempt to run swig unless you've modified the interface (*.inc, *.swig).

The python interface defines a python class for each visible Obit class in a separate *.py file. The python/Obit classes are thin objects with basically a pointer to the c structure (the "me" member) but this allows mapping the python memory usage scheme onto Obit's similar but less automatic scheme. Thus the c objects created are automatically deleted when the python reference count drops to zero or the explicit destructor (del) is invoked. However, this operation in python is only performed during its occasional garbage collection; to ensure timely deallocation of large Obit objects use the explicit Unref function. All functions are implemented as non-class member routines which take class arguments but many common functions (e.g. Open, Close, Read) are also implemented as class member functions.

The convention for the routine names on the c side of the swig interface is the same as the Obit name but dropping the initial "Obit". This adds another layer of routine call but allows translating the data types across the python/c divide. Swig (plus the type maps in swig and .inc files) helps but sometimes this is insufficient; having a uniform way of dealing with the interface improves reliability. Some of the functions defined in the *.inc files correspond to macro expansions or straight access to member values in c.

On the python side of the interface, a separate layer of routines is added although generally the swig defined c routines are available but while more efficient, direct usage may circumvent the automatic destruction feature.

Access

Functions running in the ObitTalk python session can only "see" data which is locally visible. There is limited access to data on remote machines but not to other class libraries on the remote site.

OTObit

In order to simplify the user interface and make it look more like AIPS/POPS, ObitTalk imports the OTObit module which defines many convenience functions. Asynchronous tasking is also implemented in OTObit. To run a task asynchronously, it is necessary for its messages to appear in a window other than the one in which the user is typing. These windows are implemented using the wxPython python module (where available). The functions in OTObit are described more fully in the ObitTalk user document and in on–line documentation.

1.3.5 Access to Data

Access to data, especially data descriptors is vital even if the bulk of the processing is performed in tasks. There are several ways of accessing data depending on whether or not they **may** be on a

remote machine.

Local access and manipulation

Bindings to Obit classes and objects through the swig python/c interface is relatively straightforward as long as all data is locally visible. Class objects are created in the c routines and pointers passed to python. There are interface routines which allow access to the data directly and allow access to the Obit class libraries to do major functions, e.g. imaging, deconvolving, self calibration. This "tool–kit" approach allows task–like functionality in a much more flexible form. Since only the high level control functions are in python there is little efficiency loss over tasks.

Remote or local access

Even for processing using tasks on remote machines some data access is required. For this ObitTalk provides AIPSData classes for access to data on remote machines. In this case, the remote (or local) proxy reads/writes the data and passes it back and forth over the xmlrpc interface.

1.4 Interprocess Communication

Any time multiple independent processes are used for an operation, even if they all run in the same machine, some means of communication or coordination is needed. In this age of distributed data and computing, communication over networks is needed. In ObitTalk and related software, there are two kinds of interprocess communications. Python processes which start tasks as independent processes on the same machine communicate to these processes mainly through disk files. All other interprocess communications are by means of xmlrpc which allows these to be over networks.

1.4.1 xmlrpc

Xmlrpc is a relatively simple communication technique using http protocols in a stateless query-response model. Thus, any state (e.g., one or more tasks currently active) must reside in the processes involved and activities such as processing message logs and waiting for task completion are done by polling from the client rather than interrupts or messages from the server. The implementation in Obit and related is a relatively thin layer which could be replaced by another protocol of similar capabilities.

One of the principle operational difficulties with xmlrpc is the need for preassigned port numbers; the client needs to know which port the server is watching. In theory, port 80 could be used but this requires that a web server be installed which knows how to talk to the relevant service.

An advantage of xmlrpc is its relative flexibility, since call arguments and return values are always xml strings, calls do not need to have predefined arguments and returns. Another advantage of xmlrpc is that it has widespread implementations; both python and c interfaces are implemented in Obit and related.

1.4.2 Communications with tasks

In ObitTalk, tasks are started by a python process running in the target host machine. Input and output parameters are communicated by means of a locally visible disk file. In the case of AIPS tasks, this is a single binary file per host and each "POPS" number has an assigned portion of the (TD) file. Obit tasks use independent text files for each task instance and separate files for input and output parameters. The names of these files are passed as command line arguments when the

task process is started. Prior to task initiation, the input parameters are written to the relevant files and the task process started. When the task finishes, it writes any return parameters and a code indicating whether the process was satisfied with the execution are written into the output file. The python proxy then extracts the output parameters and the completion code to return to the client. In the case of a less than completely satisfactory conclusion of a synchronous execution of the task, a RunTime python exception is thrown causing any script being executed to abort.

Communication with an executing process is more complex. AIPS and Obit tasks can interactively use their respective display processes (XAS and ObitView) to control processing. The process may also solicit and accept terminal input from the user; in this case, such communications is over the client–proxy interface. A feature of AIPS tasks which is not implemented but probably should be (and added to Obit Tasks) is the ability to pass revised parameters during task execution (SHOW/TELL in POPS). In AIPS tasks this is by means of updating parameters in the binary file and the task occasionally checking for redefined parameters. This allows users to modify the behavior during run time and in particular to inform an iterative process that its current state is sufficient and it should stop.

1.4.3 Communications with Image Displays

Image display processes run independently, usually on the client side but it is also possible to run them on the server with a X-windows display on the client. Communications with display processes can be done over networks. This allows displays to be run on a user's local (client) machine and talk to a process on a remote server. This is relatively straightforward with the AIPS display, XAS, as the image is passed over the network using a socket connection. The disadvantage is that the AIPS display has no display controls of its own and requires another process to send it commands. AIPS tasks may access the XAS display but ObitTalk has no direct capability.

Communication with ObitView is by means of xmlrpc which also allows network access. If the ObitView display server is running on a host different from the one on which the data resides (not localhost), then the image is written as a gzipped FITS image and copied to the ObitView server over the xmlrpc connection where it can load the image. FITS images may include a URL so may be accessed over a network. An additional complication is that AIPS data is stored in local data format (e.g. big vs. little endian) and remote access may not be useful. Once an image is loaded into ObitView it has many options for modifying the display or simple analysis of the image. Both Obit tasks and ObitTalk have access to ObitView.

1.5 Example Usage

The following sections give examples of the various levels of access to Obit and AIPS software from python and ObitTalk in particular.

1.5.1 ObitTalk Task Example

An example of creating a task object named im to run AIPS task IMEAN is:

```
>>> im=AIPSTask("IMEAN")
```

The parameters of the task can then be set:

```
>>> im.inname='07030+51396'; im.inclass='PCUBE'; im.indisk=1; im.inseq=2
>>> im.BLC=AIPSList([10,10]); im.TRC=AIPSList([100,100])
```

The Inputs can be reviewed:

```
>>> im.inputs()
IMEAN:  Task to print the mean, rms and extrema in an image
Adverbs     Values                                      Comments
--------------------------------------------------------------------------------
 dohist      -1.0                                       True (1.0) do histogram plot.
                                                        = 2 => flux on x axis
 userid       0.0                                       User ID.  0=>current user
                                                          32000=>all users
 inname      07030+51396                                Image name (name)
 inclass     PCUBE                                      Image name (class)
 inseq        2.0                                       Image name (seq. #)
 indisk       1.0                                       Disk drive #
 blc         10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0        Bottom left corner of image
                                                        0=>entire image
 trc         100.0, 100.0, 0.0, 0.0, 0.0, 0.0, 0.0      Top right corner of image
                                                        0=>entire image
 nboxes       0.0                                       No. of ranges for histogram.
 pixrange    0.0, 0.0                                   Min and max range for hist.
 functype                                               'LG' => do log10 plot of #
                                                        samples, else linear
 pixavg       0.0                                       Estimate of mean noise value
 pixstd       0.0                                       Estimate of true noise rms
                                                           < 0 => don't do one
                                                           = 0 => 2-passes to get
 docat        1.0                                       Put true RMS in header
 ltype        3.0                                       Type of labeling: 1 border,
                                                        2 no ticks, 3 - 6 standard,
                                                        7 - 10 only tick labels
                                                        <0 -> no date/time
 outfile                                                Name of output log file,
                                                        No output to file if blank
 dotv        -1.0                                       > 0 Do plot on the TV, else
                                                        make a plot file
 grchan       0.0                                       Graphics channel 0 => 1.
```

and the task run:

```
>>> log=im.go()
IMEAN2: Task IMEAN  (release of 31DEC02) begins
IMEAN2: Initial guess for PIXSTD taken from ACTNOISE inheader
IMEAN2: Image= 07030+51396 .PCUBE .   2 1   xywind=    1    1  241  241
IMEAN2: Mean and rms found by fitting peak in histogram:
IMEAN2: Mean=-3.1914E-06 Rms= 2.7893E-04  **** from histogram
IMEAN2: Mean and rms found by including all data:
IMEAN2: Mean= 1.8295E-05 Rms= 5.2815E-04 JY/BEAM  over    174243 pixels
IMEAN2: Flux density =  2.0006E-01 Jy.   beam area =  15.93 pixels
IMEAN2: Minimum=-1.5441E-03 at  164  180    1    1
IMEAN2: Skypos: RA 07 02 04.303  DEC 51 51 23.18
IMEAN2: Skypos: IPOL  1400.000 MHZ
```

```
IMEAN2: Maximum= 4.0180E-02 at   93  159    1    1
IMEAN2: Skypos: RA 07 03 36.211  DEC 51 47 11.65
IMEAN2: Skypos: IPOL  1400.000 MHZ
IMEAN2: returns adverbs to AIPS
IMEAN2: Appears to have ended successfully
IMEAN2: smeagle      31DEC02 TST: Cpu=       0.0  Real=       0
```

1.5.2 Obit class from ObitTalk example

Obit classes can be accessed from python scripts or interactively. A number of scripts are available in the python subdirectory with names "script*.py", most of these can be run in python without ObitTalk. Obit python functions should all have documentation strings that can be accessed interactively. The examples shown in this section only work for localy visible data. The following example from UVImager shows the help facility. Note: help also works on entire python modules.

```
>>> import UVImager
>>>  help(UVImager.UVImage)
Help on function UVImage in module UVImager:

UVImage(err, input={'Channel': 0, 'DoBeam': True, 'DoWeight': False, 'InData': None, 'OutImage
s': None, 'Robust': 0.0, 'TimeRange': [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], 'UVTaper': [0.
0, 0.0], 'WtBox': 0, 'WtFunc': 1, ...})
    Image a uv data set.

    UV Data is weighted and imaged producing an array of images.
    err     = Python Obit Error/message stack
    input   = input parameter dictionary

    Input dictionary entries:
    InData   = Input Python OTF to image
    OutImages= Output image mosaic, image objects should be previously defined
    DoBeam   = True if beams are to be made
    DoWeight = If True apply uniform weighting corrections
    Robust   = Briggs robust parameter. (AIPS definition)
    UVTaper  = UV plane taper, sigma in klambda as [u,v]
    WtSize   = Size of weighting grid in cells [same as image nx]
    WtBox    = Size of weighting box in cells [def 1]
    WtFunc   = Weighting convolution function [def. 1]
               1=Pill box, 2=linear, 3=exponential, 4=Gaussian
               if positive, function is of radius, negative in u and v.
    WtPower  = Power to raise weights to.  [def = 1.0]
               Note: a power of 0.0 sets all the output weights to 1 as modified
               by uniform/Tapering weighting.
               Applied in determing weights as well as after.
    Channel  = Channel (1-rel) number to image, 0-> all.
```

Many of the higher level functions have inputs in the form of a python dictionary whose current values can be displayed by the class input function, e.g.:

```
>>> UVImager.input(UVImager.UVImageInput)
Inputs for  UVImage
```

```
   InData     =  None  :  Input UV data
   OutImages  =  None  :  Output image mosaic
   DoBeam     =  True  :  True if beams are to be made
   DoWeight   =  False  :  If True apply uniform weighting corrections to uvdata
   Robust     =  0.0  :  Briggs robust parameter. (AIPS definition)
   UVTaper    =  [0.0, 0.0]  :  UV plane taper, sigma in klambda as [u,v]
   WtSize     =  -1  :  Size of weighting grid in cells [image]
   WtBox      =  0  :  Size of weighting box in cells [def 0]
   WtFunc     =  1  :  Weighting convolution function [def. 1]
   WtPower    =  1.0  :  Power to raise weights to.  [def = 1.0]
   Channel    =  0  :  Channel (1-rel) number to image, 0-> all.
Note: the values displayed for Obit objects are the names you give
them.
```

Obit messages and error handling is by means of the Obit type ObitErr which is an argument to most Obit routines. This contains informative as well as error messages; its contents can be displayed using e.g.:

```
>>> OErr.printErr(err)
** Message: information  : Hogbom CLEANed 300 components with 4.025047 Jy
** Message: information  : Reached minimum flux density -0.335160 Jy
** Message: information  : Scaling residuals by 1.079005
** Message: information  : Restoring 300 components
```

If the err object indicates an error, OErr.printErr(err), will raise an exception.

If scratch files are created (see OTF.ResidCal for an example) then an Obit shutdown will delete them:

```
# Shutdown Obit
OErr.printErr(err)
OSystem.Shutdown(ObitSys)
```

1.5.3 Python script without ObitTalk example

A python script that is the functional equivalent of the AIPS task HGEOM (as originally intended, interpolate the pixels of one image onto the grid defined by another) is shown in the following. The image is interpolated into a scratch image which is then copied to the output image as a quantized (1/4 RMS of noise) image , the old history is copied and new history records written. Note: This is also implemented as Obit Task HGeom. This script does many of the initializations done automatically by ObitTalk.

```
# python/Obit equivalent of AIPSish HGEOM

import Obit, Image, ImageUtil, OSystem, OErr

# Init Obit
err=OErr.OErr()

# Use default directories
user = 100
pgmNumber = 1
```

```
ObitSys=OSystem.OSystem ("HGeom", pgmNumber, user, -1, ["Def"], \
    -1, ["Def"], True, False, err)
#  print any error messages and halt
OErr.printErrMsg(err, "Error with Obit startup")

# Define files (FITS)
# Image to be interpolated onto the grid of tmplFile
inDisk = 1
inFile   = 'input.fits'
# Image defining the output grid
tmplDisk = 1
tmplFile = 'template.fits'
# output file, the '!' allows overwriting an existing file
outDisk = 1
outFile  = '!HGeomOut.fits'

# Create Python/Obit objects attached to the data files
inImage   = Image.newPFImage("Input image",    inFile,   inDisk,   1, err)
tmplImage = Image.newPFImage("Template image", tmplFile, tmplDisk, 1, err)
outImage  = Image.newPFImage("Output image",   outFile,  outDisk,  0, err)
Image.PClone(tmplImage, outImage, err)   # Same structure etc.
OErr.printErrMsg(err, "Error initializing")

# Generate scratch file from tmplFile
tmpImage  = Image.PScratch(tmplImage, err)
tmpImage.Open(Image.WRITEONLY, err)   # Open
OErr.printErrMsg(err, "Error cloning template")

# Interpolate pixels to temporary file
ImageUtil.PInterpolateImage(inImage, tmpImage, err)
OErr.printErrMsg(err, "Error interpolating")

# Do history to scratch image as table
inHistory  = History.History("history", inImage.List, err)
outHistory = History.History("history", tmpImage.List, err)
History.PCopyHeader(inHistory, outHistory, err)
# Add this programs history
outHistory.Open(History.READWRITE, err)
outHistory.TimeStamp(" Start Obit "+ObitSys.pgmName,err)
outHistory.WriteRec(-1,ObitSys.pgmName+" / input = "+inFile,err)
outHistory.WriteRec(-1,ObitSys.pgmName+" / template = "+tmplFile,err)
outHistory.Close(err)
OErr.printErrMsg(err, "Error with history")

# Copy to quantized integer image with history
print "Write output image"
inHistory  = History.History("history", tmpImage.List, err)
Image.PCopyQuantizeFITS (tmpImage, outImage, err, inHistory=inHistory)
```

```
# Say what happened
print "Interpolated",inFile,"to",outFile,"a clone of ",tmplFile

# Shutdown Obit
OErr.printErr(err)  # Print any remaining Obit messages
OSystem.Shutdown(ObitSys)
```

1.6 ObitTalk Implementation

The following are the internal documentations of major ObitTalk classes.

1.6.1 Client side

The client side software always runs in the machine the user is running the python process on.

ObitTask

The client ObitTask class provides a means to define the parameters, run, receive messages and receive return values for Obit tasks.

```
class ObitTask(AIPSTask.AIPSTask)
This class implements running Obit tasks.

The ObitTask class, derived from the AIPSTask class, handles
client-side task related operations.  Actual task definition
and operations are handled by server-side proxies.
For local operations, the server-side functionality is implemented
in the same address space but remote operation is through an
xmlrpc interface.  Tasks are run as separate processes in all
cases.

Each defined disk has an associated proxy, either local or remote.
A proxy is a module with interface functions,
local proxies are class modules from subdirectory Proxy with the
same name (i.e. AIPSTask) and the server functions are implemented
there.  Remote proxies are specified by a URL and a proxy from the
xmlrpclib module is used.

When an object is created, the task secific parameters and
documentation are retrieved by parsing the task TDF file.
This is performed on the server-side.

Method resolution order:
    ObitTask
    AIPSTask.AIPSTask
    Task.Task
    MinimalMatch.MinimalMatch

Methods defined here:
```

```
__init__(self, name)
    Create Obit task object

    Creates task object and calls server function to parse TDF
    file to obtain task specific parametrs and documentation.
    Following is a list of class members:
    _default_dict   = Dictionary with default values of parameters
    _input_list     = List of input parameters in order
    _output_list    = List of output parameters in order
    _min_dict       = Parameter minimum values as a List
    _max_dict       = Parameter maximum values as a List
    _hlp_dict       = Parameter descriptions (list of strings)
                      as a dictionary
    _strlen_dict    = String parameter lengths as dictionary
    _help_string    = Task Help documentation as list of strings
    _explain_string = Task Explain documentation as list of strings
    _short_help     = One line description of task
    _message_list   = list of execution messages
    retCode         = Task return code, 0=Finished OK
    debug           = If true save task parameter file
    logFile         = if given, the name of the file in which to
                      write messages
    doWait          = True if synchronous  operation wished
    Current parameter values are given as class members.

abort(self, proxy, tid, sig=9)
    Abort the task specified by PROXY and TID.

    Calls abort function for task tid on proxy.
    None return value
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process to be terminated
    sig   = signal to sent to the task

go(self)
    Run the task.

    Writes task input parameters, data directories and other
    information in the task parameter file and starts the task
    synchronously returning only when the task terminates.
    Messages are displayed as generated by the task,
    saved in an array returned from the call and, if the task
    member logFile is set, written to this file.

messages(self, proxy=None, tid=None)
    Return messages for the task specified by PROXY and TID.
```

```
    Returns list of messages and appends them to the object's
    message list.
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

spawn(self)
    Spawn the task.

    Writes task input parameters, data directories and other
    information in the task parameter file and starts the task
    asynchronously returning immediately.  Messages must be
    retrieved calling messages.
    Returns (proxy, tid)

----------------------------------------------------------------------
Data and other attributes defined here:

debug = False

doWait = False

logFile = ''

version = 'OBIT'

----------------------------------------------------------------------
Methods inherited from AIPSTask.AIPSTask:

__call__(self)

__eq__(self, other)
    Check if two task objects are for the same task

__getattr__(self, name)

__setattr__(self, name, value)

copy(self)
    Return a copy of a given task object

defaults(self)
    Set adverbs to their defaults.

feed(self, proxy, tid, banana)
    Feed the task a  BANANA.

    Pass a message to a running task's sdtin
    proxy   = Proxy giving access to server
```

```
    tid     = Task id in pid table of process
    bananna = text message to pass to task input

finished(self, proxy, tid)
    Determine if task has finished

    Determine whether the task specified by PROXY and TID has
    finished.
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

inputs(self)
    Display all inputs for this task.

outputs(self)
    Display all outputs for this task.

wait(self, proxy, tid)
    Wait for the task to finish.

    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

----------------------------------------------------------------------
Data and other attributes inherited from AIPSTask.AIPSTask:

isbatch = 0

msgkill = 0

userno = 0

----------------------------------------------------------------------
Methods inherited from Task.Task:

explain(self)
    Display help+explain for this task.

help(self)
    Display help for this task.

----------------------------------------------------------------------
Methods inherited from MinimalMatch.MinimalMatch:

__repr__(self)
```

AIPSTask

AIPSTask objects allow means to define the parameters, run, receive messages and receive return values for AIPS tasks.

```
class AIPSTask(Task.Task)
This class implements running AIPS tasks.

The AIPSTask class, handles client-side task related operations.
Actual task definition and operations are handled by server-side
proxies. For local operations, the server-side functionality is
implemented in the same address space but remote operation is
through an xmlrpc interface.  Tasks are run as separate processes
in all cases.

Each defined disk has an associated proxy, either local or remote.
A proxy is a module with interface functions,
local proxies are class modules from subdirectory Proxy with the
same name (i.e. ObitTask) and the server functions are implemented
there.  Remote proxies are specified by a URL and a proxy from the
xmlrpclib module is used.

When an object is created, the task secific parameters and
documentation are retrieved by parsing the task Help file.and the
POPSDAT.HLP file for parameter definitions.  This is performed on
the server-side.

Method resolution order:
    AIPSTask
    Task.Task
    MinimalMatch.MinimalMatch

Methods defined here:

__call__(self)

__eq__(self, other)
    Check if two task objects are for the same task

__getattr__(self, name)

__init__(self, name, **kwds)
    Create AIPS task object

    Creates task object and calls server function to parse the
    task help and POPSDAT.HLP files to obtain task specific
    parametrs and documentation.
    Following is a list of class members:
    _default_dict   = Dictionary with default values of parameters
```

```
    _input_list     = List of input parameters in order
    _output_list    = List of output parameters in order
    _min_dict       = Parameter minimum values as a List
    _max_dict       = Parameter maximum values as a List
    _hlp_dict       = Parameter descriptions (list of strings)
                      as a dictionary
    _strlen_dict    = String parameter lengths as dictionary
    _help_string    = Task Help documentation as list of strings
    _explain_string = Task Explain documentation as list of strings
    _short_help     = One line description of task
    _message_list   = list of execution messages
    Current parameter values are given as class members.

__setattr__(self, name, value)

abort(self, proxy, tid, sig=15)
    Abort the task specified by PROXY and TID.

    Calls abort function for task tid on proxy.
    None return value
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process to be terminated
    sig   = signal to sent to the task

copy(self)
    Return a copy of a given task object

defaults(self)
    Set adverbs to their defaults.

feed(self, proxy, tid, banana)
    Feed the task a  BANANA.

    Pass a message to a running task's sdtin
    proxy   = Proxy giving access to server
    tid     = Task id in pid table of process
    bananna = text message to pass to task input

finished(self, proxy, tid)
    Determine if task has finished

    Determine whether the task specified by PROXY and TID has
    finished.
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

go(self)
    Run the task.
```

```
    Writes task input parameters in the task parameter file and
    starts the task synchronously returning only when the task
    terminates. Messages are displayed as generated by the task,
    saved in an array returned from the call and, if the task
    member logFile is set, written to this file.

inputs(self)
    Display all inputs for this task.

messages(self, proxy=None, tid=None)
    Return task messages

    Returns list of messages and appends them to the object's
    message list.
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

outputs(self)
    Display all outputs for this task.

spawn(self)
    Spawn the task.

    Writes task input parameters, task parameter file and starts
    the task asynchronously returning immediately.  Messages must be
    retrieved calling messages.
    Returns (proxy, tid)

wait(self, proxy, tid)
    Wait for the task to finish.

    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

----------------------------------------------------------------------
Data and other attributes defined here:

doWait = False

isbatch = 0

logFile = ''

msgkill = 0

userno = 0
```

```
version = 'OLD'

----------------------------------------------------------------------
Methods inherited from Task.Task:

explain(self)
    Display help+explain for this task.

help(self)
    Display help for this task.

----------------------------------------------------------------------
Methods inherited from MinimalMatch.MinimalMatch:

__repr__(self)
```

ObitScript

This is the client interface to local or remote script execution.

```
DESCRIPTION
    This module provides the ObitScript class.
    This class allows running Obit/python scripts either
    locally or remotely

    ObitScripts are derived from Task and share most of execution properties.
    In particular, ObitScripts can be executed either locally or remotely.
    In this context a script is a character string containing a sequence of
    ObitTalk or other python commands and may be included when the script
    object is created or attached later.
    An example:
    script="import OSystem
    print 'Welcome user',OSystem.PGetAIPSuser()
    "

CLASSES
    ObitScriptMessageLog
    Task.Task(MinimalMatch.MinimalMatch)
        ObitScript

  class ObitScript(Task.Task)
     This class implements running Obit/python Script

     The ObitScript class, handles client-side script related operations.
     Actual script operations are handled by server-side proxies.
     For local operations, the server-side functionality is
     implemented in the same address space but remote operation is
     through an xmlrpc interface.
```

```
An ObitScript has an associated proxy, either local or remote.
A proxy is a module with interface functions,
local proxies are class modules from subdirectory Proxy with the
same name (i.e. ObitScript) and the server functions are implemented
there.  Remote proxies are specified by a URL and a proxy from the
xmlrpclib module is used.

Method resolution order:
    ObitScript
    Task.Task
    MinimalMatch.MinimalMatch

Methods defined here:

__call__(self)

__getattr__(self, name)

__init__(self, name, **kwds)
    Create ObitScript task object

    Creates Script Object.
    name  = name of script object
    Optional Keywords:
        script   = Script to execute as string
        URL      = URL on which the script is to be executed
                   Default = None = local execution
        AIPSDirs = List of AIPS directories on URL
                   Default = current AIPS directories on url
        FITSDirs = List of FITS directories on URL
                   Default = current FITS directories on url
        AIPSUser = AIPS user number for AIPS data files
                   Default is current
        version  = AIPS version string, Default = current
    Following is a list of class members:
        url      = URL of execution server, None=Local
        proxy    = Proxy for URL
        script   = Script as text string
        userno   = AIPS user number
        AIPSDirs = List of AIPS directories on URL
        FITSDirs = List of FITS directories on URL
        AIPSUser = AIPS user number for AIPS data files
        version  = AIPS version string
        _message_list = messages from Script execution

__setattr__(self, name, value)

abort(self, proxy, tid, sig=15)
```

```
    Abort the script specified by PROXY and TID.

    Calls abort function for task tid on proxy.
    None return value
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process to be terminated
    sig   = signal to sent to the task

explain(self)
    List script

feed(self, proxy, tid, banana)
    Feed the script a  BANANA.

    Pass a message to a running script's sdtin
    proxy   = Proxy giving access to server
    tid     = Script task id in pid table of process
    bananna = text message to pass to script input

finished(self, proxy, tid)
    Determine if script has finished

    Determine whether the script specified by PROXY and TID has
    finished.
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

go(self)
    Execute the script.

    Writes task input parameters in the task parameter file and
    starts the task synchronously returning only when the task
    terminates. Messages are displayed as generated by the task,
    saved in an array returned from the call and, if the task
    member logFile is set, written to this file.

help(self)
    List script.

inputs(self)
    List script

messages(self, proxy=None, tid=None)
    Return task messages

    Returns list of messages and appends them to the object's
    message list.
    proxy = Proxy giving access to server
```

```
    tid   = Task id in pid table of process

outputs(self)
    Not defined.

spawn(self)
    Spawn the script.

    Starts script asynchronously returning immediately
    Messages must be retrieved calling messages.
    Returns (proxy, tid)

wait(self, proxy, tid)
    Wait for the script to finish.

    proxy = Proxy giving access to server
    tid   = Task id in pid table of process

----------------------------------------------------------------------
Data and other attributes defined here:

AIPSDirs = []

FITSDirs = []

debug = False

doWait = False

isbatch = 32000

logFile = ''

msgkill = 0

proxy = <module 'LocalProxy' from '/export/users/bcotton/share/obittal...

script = ''

url = None

userno = 0

version = 'TST'

----------------------------------------------------------------------
Methods inherited from MinimalMatch.MinimalMatch:
```

```
    __repr__(self)

  class ObitScriptMessageLog
    Methods defined here:

    __init__(self)

    zap(self)
        Zap message log.

    ----------------------------------------------------------------------
    Data and other attributes defined here:

    userno = -1
```

AIPSData

This is the client interface to local or remote AIPS data.

```
Help on module AIPSData:

NAME
    AIPSData

FILE
    /export/users/bcotton/share/obittalk/python/AIPSData.py

DESCRIPTION
    This module provides the AIPSImage and AIPSUVData classes.  These
    classes implement most of the data oriented verb-like functionality
    from classic AIPS.

CLASSES
    AIPSCat
    _AIPSData
        AIPSImage
        AIPSUVData

    class AIPSCat
    Methods defined here:

    __init__(self, disk)

    __repr__(self)

    class AIPSImage(_AIPSData)
    This class describes an AIPS image.

    Methods inherited from _AIPSData:
```

```
__getattr__(self, name)

__init__(self, name, klass, disk, seq)

__repr__(self)

__str__(self)

exists(self)
    Check whether this image or data set exists.

    Returns True if the image or data set exists, False otherwise.

getrow_table(self, type, version, rowno)
    Get a row from an extension table.

    Returns row ROWNO from version VERSION of extension table TYPE
    as a dictionary.

header(self)
    Get the header for this image or data set.

    Returns the header as a dictionary.

header_table(self, type, version)
    Get the header of an extension table.

    Returns the header of version VERSION of the extension table
    TYPE.

table(self, type, version)

table_highver(self, type)
    Get the highest version of an extension table.

    Returns the highest available version number of the extension
    table TYPE.

tables(self)
    Get the list of extension tables.

verify(self)
    Verify whether this image or data set can be accessed.

zap(self)
    Destroy this image or data set.
```

```
zap_table(self, type, version)
    Destroy an extension table.

    Deletes version VERSION of the extension table TYPE.  If
    VERSION is 0, delete the highest version of table TYPE.  If
    VERSION is -1, delete all versions of table TYPE.

----------------------------------------------------------------------
Properties inherited from _AIPSData:

disk
    Disk where this data set is stored.

    lambdaself

klass
    Class of this data set.

    lambdaself

name
    Name of this data set.

    lambdaself

seq
    Sequence number of this data set.

    lambdaself

userno
    User number used to access this data set.

    lambdaself

class AIPSUVData(_AIPSData)
This class describes an AIPS UV data set.

Methods inherited from _AIPSData:

__getattr__(self, name)

__init__(self, name, klass, disk, seq)

__repr__(self)

__str__(self)
```

```
exists(self)
    Check whether this image or data set exists.

    Returns True if the image or data set exists, False otherwise.

getrow_table(self, type, version, rowno)
    Get a row from an extension table.

    Returns row ROWNO from version VERSION of extension table TYPE
    as a dictionary.

header(self)
    Get the header for this image or data set.

    Returns the header as a dictionary.

header_table(self, type, version)
    Get the header of an extension table.

    Returns the header of version VERSION of the extension table
    TYPE.

table(self, type, version)

table_highver(self, type)
    Get the highest version of an extension table.

    Returns the highest available version number of the extension
    table TYPE.

tables(self)
    Get the list of extension tables.

verify(self)
    Verify whether this image or data set can be accessed.

zap(self)
    Destroy this image or data set.

zap_table(self, type, version)
    Destroy an extension table.

    Deletes version VERSION of the extension table TYPE.  If
    VERSION is 0, delete the highest version of table TYPE.  If
    VERSION is -1, delete all versions of table TYPE.

----------------------------------------------------------------------
Properties inherited from _AIPSData:
```

```
    disk
        Disk where this data set is stored.

        lambdaself

    klass
        Class of this data set.

        lambdaself

    name
        Name of this data set.

        lambdaself

    seq
        Sequence number of this data set.

        lambdaself

    userno
        User number used to access this data set.

        lambdaself
```

FITSData

This is the client interface to local FITS data. There appears to be no server side equivalent.

```
Help on module FITSData:

NAME
    FITSData

FILE
    /export/users/bcotton/share/obittalk/python/FITSData.py

DESCRIPTION
    This module provides the FITSImage and FITSUVData classes.  These
    classes implement most of the data oriented verb-like functionality
    from classic FITS.

CLASSES
    _FITSData
        FITSImage
        FITSUVData

    class FITSImage(_FITSData)
```

```
This class describes an FITS image.

Methods inherited from _FITSData:

__getattr__(self, filename)

__init__(self, name, disk)

__repr__(self)

__str__(self)

exists(self)
    Check whether this image or data set exists.

    Returns True if the image or data set exists, False otherwise.

getrow_table(self, type, version, rowno)
    Get a row from an extension table.

    Returns row ROWNO from version VERSION of extension table TYPE
    as a dictionary.

header(self)
    Get the header for this image or data set.

    Returns the header as a dictionary.

header_table(self, type, version)
    Get the header of an extension table.

    Returns the header of version VERSION of the extension table
    TYPE.

table(self, type, version)

table_highver(self, type)
    Get the highest version of an extension table.

    Returns the highest available version number of the extension
    table TYPE.

tables(self)
    Get the list of extension tables.

verify(self)
    Verify whether this image or data set can be accessed.
```

```
zap(self)
    Destroy this image or data set.

zap_table(self, type, version)
    Destroy an extension table.

    Deletes version VERSION of the extension table TYPE.  If
    VERSION is 0, delete the highest version of table TYPE.  If
    VERSION is -1, delete all versions of table TYPE.

----------------------------------------------------------------------
Properties inherited from _FITSData:

disk
    Disk where this data set is stored.

    lambdaself

filename
    Filename of this data set.

    lambdaself

class FITSUVData(_FITSData)
This class describes an FITS UV data set.

Methods inherited from _FITSData:

__getattr__(self, filename)

__init__(self, name, disk)

__repr__(self)

__str__(self)

exists(self)
    Check whether this image or data set exists.

    Returns True if the image or data set exists, False otherwise.

getrow_table(self, type, version, rowno)
    Get a row from an extension table.

    Returns row ROWNO from version VERSION of extension table TYPE
    as a dictionary.

header(self)
```

```
    Get the header for this image or data set.

    Returns the header as a dictionary.

header_table(self, type, version)
    Get the header of an extension table.

    Returns the header of version VERSION of the extension table
    TYPE.

table(self, type, version)

table_highver(self, type)
    Get the highest version of an extension table.

    Returns the highest available version number of the extension
    table TYPE.

tables(self)
    Get the list of extension tables.

verify(self)
    Verify whether this image or data set can be accessed.

zap(self)
    Destroy this image or data set.

zap_table(self, type, version)
    Destroy an extension table.

    Deletes version VERSION of the extension table TYPE.  If
    VERSION is 0, delete the highest version of table TYPE.  If
    VERSION is -1, delete all versions of table TYPE.

----------------------------------------------------------------------
Properties inherited from _FITSData:

disk
    Disk where this data set is stored.

    lambdaself

filename
    Filename of this data set.

    lambdaself
```

1.6.2 Server side (proxy)

The following are implemented in the proxy (remote or local).

ObitTask

The server side ObitTask class performs the functions of parsing the Task definition (TDF) file for the parameters and documentation as well as operations involving the task execution.

```
Help on module Proxy.ObitTask in Proxy:

NAME
    Proxy.ObitTask - Obit tasking interface

FILE
    /export/users/bcotton/share/obittalk/python/Proxy/ObitTask.py

DESCRIPTION
    This module contains classes useful for an Obit tasking interface to python.
    An ObitTask object contains input parameters for a given Obit program.
       The parameters for a given task are defined in a Task Definition File
    (TDF) which gives the order, names, types, ranges and dimensionalities.
    A TDF is patterened after AIPS HELP files.
       The Task Definition File can be derived from the AIPS Help file with the
    addition of:
     - A line before the beginning of each parameter definition of the form:
     **PARAM** [type] [dim] **DEF** [default]
         where [type] is float or str (string) and [dim] is the
         dimensionality as a blank separated list of integers, e.g.
         **PARAM** str 12 5       (5 strings of 12 characters)
         default (optional) is the default value
     HINT: No matter what POPS thinks, all strings are multiples of 4 characters
     For non AIPS usage dbl (double), int (integer=long), boo (boolean
     "T" or "F") are defined.

CLASSES
    Proxy.Task.Task
        ObitTask

    class ObitTask(Proxy.Task.Task)
    Server-side Obit task interface

    Methods defined here:

    __init__(self)

    messages(self, tid)
        Return task's messages.
```

```
    Return a list of messages each as a tuple (1, message)
    tid   = Task id in pid table of process

params(self, name, version)
    Return parameter set for version VERSION of task NAME.

spawn(self, name, version, userno, msgkill, isbatch, input_dict)
    Start the task.

    Writes task input parameters, data directories and other
    information in the task parameter file and starts the task
    asynchronously returning immediately.  Messages must be
    retrieved calling messages.
    If the debug parameter is True then a "Dbg" copy of the task
    input file is created which will not br automatically destroyed.
    name        task name
    version     version of task
    userno      AIPS user number
    msgkill     AIPS msgkill level, not used in Obit tasks
    isbatch     True if this is a batch process , not used in Obit tasks
    input_dict  Input parameters as dictionary
    Returns task id

wait(self, tid)
    Wait for the task to finish.

    Waits for Obit task to finishes, reads the task output
    parameter file and deleted tas input and output parameter
    file.
    Returns output parameters in adictionary.
    tid   = Task id in pid table of process

----------------------------------------------------------------------
Methods inherited from Proxy.Task.Task:

abort(self, tid, sig=9)
    Abort the task

    Calls abort function for task tid
    None return value
    tid   = Task id in pid table of process to be terminated
    sig   = signal to sent to the task

feed(self, tid, banana)
    Feed the task a  BANANA.

    Pass a message to a running task's sdtin
    tid     = Task id in pid table of process
```

```
        bananna = text message to pass to task input

    finished(self, tid)
        Check whether the task has finished.

        tid   = Task id in pid table of process
```

AIPSTask

The Server side AIPSTask class performs the functions of parsing the Task help file for the parameters and the POPSDAT.HLP file to get their definitions and documentation as well as operations involving the task execution.

```
Help on module Proxy.AIPSTask in Proxy:

NAME
    Proxy.AIPSTask

FILE
    /export/users/bcotton/share/obittalk/python/Proxy/AIPSTask.py

DESCRIPTION
    This module provides the bits and pieces to implement an AIPSTask
    proxy object.

CLASSES
    AIPSMessageLog
    Proxy.Task.Task
        AIPSTask

    class AIPSMessageLog
    Methods defined here:

    __init__(self)

    zap(self, userno)
        Zap message log.

    class AIPSTask(Proxy.Task.Task)
    Server-side AIPS task interface

    Methods defined here:

    __init__(self)

    abort(self, tid, sig=15)
        Abort the task specified by PROXY and TID.

        Calls abort function for task tid on proxy.
```

```
    None return value
    proxy = Proxy giving access to server
    tid   = Task id in pid table of process to be terminated
    sig   = signal to sent to the task
            AIPS seems to ignore SIGINT, so use SIGTERM instead.

messages(self, tid)
    Return task's messages.

    Return a list of messages each as a tuple (1, message)
    tid   = Task id in pid table of process

params(self, name, version)
    Return parameter set for version VERSION of task NAME.

spawn(self, name, version, userno, msgkill, isbatch, input_dict)
    Start the task.

    Writes task input parameters in theTD file and starts the task
    asynchronously returning immediately.
    Messages must be  retrieved calling messages.
    Attempts to use singel hardcoded AIPS TV
    name        task name
    version     version of task
    userno      AIPS user number
    msgkill     AIPS msgkill level,
    isbatch     True if this is a batch process
    input_dict  Input parameters as dictionary
    Returns task id

wait(self, tid)
    Wait for the task to finish.

    When task returns, the output parameters are parser from the
    TD file.
    Returns output parameters in adictionary.
    tid   = Task id in pid table of process

----------------------------------------------------------------------
Methods inherited from Proxy.Task.Task:

feed(self, tid, banana)
    Feed the task a  BANANA.

    Pass a message to a running task's sdtin
    tid     = Task id in pid table of process
    bananna = text message to pass to task input
```

```
    finished(self, tid)
        Check whether the task has finished.

        tid   = Task id in pid table of process
```

ObitScript

This is the server interface to local or remote script execution.

```
DESCRIPTION
    This module provides the bits and pieces to implement an ObitScript
    proxy object.

CLASSES
    Proxy.Task.Task
        ObitScript

    class ObitScript(Proxy.Task.Task)
      Server-side ObitScript script interface

      Methods defined here:

      __init__(self)

      abort(self, tid, sig=15)
          Abort the script specified by PROXY and TID.

          Calls abort function for script tid on proxy.
          No return value
          proxy = Proxy giving access to server
          tid   = Script id in pid table of process to be terminated
          sig   = signal to sent to the script
                  ObitScript seems to ignore SIGINT, so use SIGTERM instead.

      messages(self, tid)
          Return script's messages.

          Return a list of messages each as a tuple (1, message)
          tid   = Script id in pid table of process

      spawn(self, name, version, userno, msgkill, isbatch, input_dict)
          Start the script.in an externak ObitTalk

          Writes script test into temporary file in /tmp and executes ObitTalk
          asynchronously returning immediately.
          Messages must be  retrieved calling messages.
          name        script name
          version     version of any AIPS tasks
```

```
        userno      AIPS user number
        msgkill     AIPStask msgkill level,
        isbatch     True if this is a batch process
        input_dict  Input info as dictionary
        Returns script id

    wait(self, tid)
        Wait for the script to finish.

        Waits until script is finished
        tid   = Script id in pid table of process

    ----------------------------------------------------------------------
    Methods inherited from Proxy.Task.Task:

    feed(self, tid, banana)
        Feed the task a  BANANA.

        Pass a message to a running task's sdtin
        tid     = Task id in pid table of process
        bananna = text message to pass to task input

    finished(self, tid)
        Check whether the task has finished.

        tid   = Task id in pid table of process
```

AIPSData

Remote data access functions live in the proxy AIPSData class.

```
Help on module Proxy.AIPSData in Proxy:

NAME
    Proxy.AIPSData

FILE
    /export/users/bcotton/share/obittalk/python/Proxy/AIPSData.py

DESCRIPTION
    This module provides the bits and pieces to implement AIPSImage and
    AIPSUVData objects.

CLASSES
    AIPSCat
    AIPSData
        AIPSImage
        AIPSUVData
```

```
class AIPSCat
Methods defined here:

__init__(self)

cat(self, disk, userno)

class AIPSData
Methods defined here:

__init__(self)

exists(self, desc)

getrow_table(self, desc, type, version, rowno)

header(self, desc)

header_table(self, desc, type, version)

table_highver(self, desc, type)

tables(self, desc)

verify(self, desc)

zap(self, desc)

zap_table(self, desc, type, version)

class AIPSImage(AIPSData)
Methods inherited from AIPSData:

__init__(self)

exists(self, desc)

getrow_table(self, desc, type, version, rowno)

header(self, desc)

header_table(self, desc, type, version)

table_highver(self, desc, type)

tables(self, desc)

verify(self, desc)
```

```
zap(self, desc)

zap_table(self, desc, type, version)

class AIPSUVData(AIPSData)
Methods inherited from AIPSData:

__init__(self)

exists(self, desc)

getrow_table(self, desc, type, version, rowno)

header(self, desc)

header_table(self, desc, type, version)

table_highver(self, desc, type)

tables(self, desc)

verify(self, desc)

zap(self, desc)

zap_table(self, desc, type, version)
```