The Codex ParselTongue

Mark Kettenis

Mike Sipior

Revision History

Revision 1.0 7 June 2007

Documents ParselTongue 1.0.6

Revision 1.0.1 7 December 2007

Added section describing the use of external tasks.

Revision 1.1 9 July 2008

Updated for ParselTongue version 1.1, adding sections on Parallel despatching and per-task log files.

Revision 1.1.1 28 November 2008

Changes to ParallelTask syntax, for consistency with AIPSTask objects. Also changed the syntax for per-task log files, again for the sake of consistency.

Table of Contents

Preliminaries 1
- Introduction 1
- Prerequisites 2
- Building Obit 4
- Building ParselTongue 4

Using ParselTongue 5
- Interactive use 5
- Scripting 6
- AIPS interface overview 6
- AIPSTasks 7
- AIPSData 9

Advanced ParselTongue 12
- Manipulating data and tables directly 12
- Moving data from one AIPS installation to another 13
- Using external tasks with ParselTongue 13
- Remote ParselTongue execution 14
- Parallel ParselTongue execution 15
- Logging 16

ParselTongue reference 17

Preliminaries

Introduction

This guide is intended as the definitive reference for the ParselTongue scripting environment. It builds upon a number of previous sources, including the *ParselTongue Tutorial* [http://www.radionet-eu.org/rnwiki/ParselTongue?action=AttachFile&do=get&target=tutorial.html] and the *ParselTongue Cookbook* [http://www.radionet-eu.org/rnwiki/ParselTongue?action=AttachFile&do=get&target=cookbook.html].

Prerequisites

ParselTongue should run on just about any modern Linux or UNIX installation. There are only three direct dependencies for a successful ParselTongue install:

- Python version > 2.2
- A working Classic AIPS environment.
- A working Obit build.

Obit [http://www.cv.nrao.edu/~bcotton/Obit.html] has its own extensive dependency list, and is generally the biggest obstacle to installing ParselTongue. In addition to a C compiler, like gcc [http://gcc.gnu.org/], the two other principal dependencies are on GLib 2.x [http://www.gtk.org/], and the GSL [http://www.gnu.org/software/gsl/] (GNU Scientific Library). If you'd like to read FITS files directly from ParselTongue, you'll also need CFITSIO [http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html]. Finally, visibility data access from ParselTongue requires either numarray [http://www.stsci.edu/resources/software_hardware/numarray], from STScI, or the newer NumPy [http://numpy.scipy.org]. NumPy is to be preferred for new projects, as numarray is now deprecated, with all development halted, including bug fixes. Optionally, FFTW, PGPLOT, XMLRPC and SWIG may also be added, although these are not necessary for building ParselTongue. In particular, we encourage folks to be careful if installing SWIG, as it is only needed if you intend to change or extend Obit itself. Obit is rather picky about the version of SWIG it requires, however, and will fail to compile with some recent versions.

Now, some more particular advice for setting up ParselTongue under the two principal types of Linux distributions, along with Apple OS X.

RPM-based distributions (SuSE, Redhat, Fedora)

You'll want to grab the following packages:

- gcc
- glib2
- glib2-devel
- gsl
- gsl-devel
- python
- python-devel
- python-numarray
- cfitsio

To figure out what packages are already on your system, try the following:

```
$ rpm -q -a
```

You'll want to pipe that through **grep**, to reduce the wear and tear on your eyeballs...

Debian, Ubuntu, and their derivatives

The package list for .deb-based systems:

- gcc
- libglib2.0-0
- libglib2.0-dev
- libgsl0
- libgsl0-dev
- python
- python-dev
- python-numarray
- libcfitsio2
- libcfitsio-dev

These should all be available in Debian main, and in the Ubuntu main and universe repositories. To see what is already installed, use **dpkg**:

```
$ dpkg -l
```

Again, piping to **grep** is advisable here.

Apple OS X

First, you'll need XCode, a developer package which ships with OS X. Whilst a part of OS X, it is generally not installed by default. Have a look on the installation DVD that came with your machine, it should be lurking there.

There are two main choices for managing third-party packages under OS X, those being Fink [http://finkproject.org/] and MacPorts [http://www.macports.org/] (formerly DarwinPorts). With Fink, you'll need the following:

- glib2
- glib2-dev
- gsl

And with MacPorts:

- glib2
- gsl

Sensing a pattern here? One tricky bit is that Apple ships a version of python with OS X. This version should work with ParselTongue, but if you decide to install numarray for the extra functionality, you'll have to build it from the source code at the STScI site. If this is a bit too much bother, install the numarray package using Fink/MacPorts, and it should install a (probably more recent) version of python *en passant*.

You'll need to make sure that you invoke the correct version, and so will need to set the PYTHON environmental variable appropriately. For Fink, using the **bash** shell, the following will do:

```
$ export PYTHON=/sw/bin/python
```

For **csh**/**tcsh**, try:

```
% setenv PYTHON /sw/bin/python
```

If you've used MacPorts instead, the appropriate value for PYTHON is /opt/local/bin/python.

Building Obit

After the package gymnastics above, building Obit should be relatively anti-climactic. Here [http://www.radionet-eu.org/rnwiki/ParselTongue?action=AttachFile&do=get&target=Obit-20070702.tar.gz] you can find a tarball for a modified version of Obit (20070702) that is known to work with ParselTongue. After downloading, simply extract the tarball, configure, and make:

```
$ tar zxvf Obit-20070702.tar.gz
$ cd Obit
$ ./configure
$ make
```

If all dependencies have been satisfied, this should build with no trouble.

Building ParselTongue

Warning

When upgrading to a new version of ParselTongue, be sure to remove any .ParselTongue directory found in your home directory. This directory contains cached information on AIPS tasks, and the format of this may change between versions of ParselTongue.

```
$ rm -rf ~/.ParselTongue
```

You can find the latest version of ParselTongue here [http://www.jive.nl/parseltongue/releases/parseltongue.tar.gz]. The only bit of information you will require during the install is the location of the Obit build you created in the previous step, which is passed to **configure**:

```
$ tar zxvf parseltongue-1.1.1.tar.gz
$ cd parseltongue-1.1.1
$ ./configure --with-obit=/path/to/Obit
$ make
$ sudo make install
```

Warning

To install with NumPy functionality, simply use the --with-numpy flag for **configure**. To switch back to a numarray-based installation of ParselTongue, simply replace the file $PTROOT /python/

Wizardry/AIPSData.py with the file $PTROOT/python/Wizardry/AIPSData.py.orig, and rerun **make** and **make install**.

As of version 1.1, ParselTongue still defaults to using numarray, but this will almost certainly change in future releases.

The last step must be performed either via sudo, or logged in directly as root. If you don't have superuser privileges on your machine, you can specify an installation directory different from the default (/usr/local) with the "--prefix" option to **configure**:

```
$ ./configure --prefix=$prefix --with-obit=/path/to/Obit
```

If you installed ParselTongue in the default location, you should be all set at this point (/usr/local/bin should already be in your path). If you changed the default install location, you should remember to add this directory to your path as well:

```
$ export PATH=$PATH:$prefix/bin
```

or:

```
% setenv PATH $PATH:$prefix/bin
```

Using ParselTongue

Before using ParselTongue, ensure that your AIPS environment is loaded properly by sourcing the appropriate AIPS login script. Assuming the root of the AIPS install on your system is located at the directory $AIPSROOT, the relevant command for the Bourne shell and its descendents (**sh**, **bash**, **ksh** *et alia*) is:

```
$ . $AIPSROOT/LOGIN.SH
```

For **csh** and **tcsh**, try:

```
% source $AIPSROOT/LOGIN.CSH
```

Interactive use

Whilst primarily intended for scripting, ParselTongue is perfectly suitable for interactive use. After invoking the **ParselTongue**, you will be prompted for your AIPS user number, and you will get back a standard Python prompt:

```
$ ParselTongue
Python 2.5.1c1 (release25-maint, Apr 12 2007, 21:00:25)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

Welcome to ParselTongue 1.0.5
Please enter your AIPS user ID number: 667
>>>
```

The first time you do this, ParselTongue will take a few moments to respond, as it goes over your AIPS installation to see which tasks and adverbs are available.

User 667 is, of course, the Neighbour of the Beast.

Scripting

ParselTongue can be used to run your scripts directly:

```
$ ParselTongue script.py
```

This sets up the required AIPS environment, but to use ParselTongue from your scripts, you'll still need to import the appropriate module set. Here's a set that should get you started:

```
from AIPS import AIPS
from AIPSTask import AIPSTask, AIPSList
from AIPSData import AIPSUVData, AIPSImage
```

You can add more modules as you need them, and can of course import any other Python module you might require. One difference between running ParselTongue interactively and running a ParselTongue script is that your AIPS user ID will not be set unless your script explicitly does so with the following:

```
AIPS.userno=667
```

AIPS interface overview

ParselTongue exposes a large subset of the AIPS interface to your scripts, including every AIPS Task. Whilst AIPS Verbs are not directly supported, their functionality can usually be mimicked with snippets of Python code. Under ParselTongue, an AIPS Task is represented by an object, and the attributes of that object correspond to the "adverbs" AIPS uses for parameter passing. For example, to load a FITS file into the AIPS catalogue, we can make use of the **FITLD** task as follows:

```
$ ParselTongue
Python 2.5.1c1 (release25-maint, Apr 12 2007, 21:00:25)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

Welcome to ParselTongue 1.0.5
Please enter your AIPS user ID number: 667
>>> fitld = AIPSTask('FITLD') # make a new AIPSTask object
>>> fitld.infile = "/home/sipior/data/N1066.fits"
>>> fitld.outname = "N1066"   # set output name in the AIPS catalogue
>>> fitld.go()                # despatch the job
```

Note that the choice of "fitld" as the object name is entirely arbitrary, and is intended as a mnemonic. More importantly (especially to you old AIPS hands), note that each instance of an AIPSTask has its own private set of adverbs! This allows for much greater flexibility than POPS, but be aware that these must be explicitly set. N.B. All adverb names are in the lower-case!

Data sets that have been loaded into the AIPS catalogue can be accessed as AIPSData objects, which we will discuss shortly in the section AIPSData, below. Once an AIPSData object is associated with a data set, that object can be used directly as an adverb in other AIPS Tasks.

AIPSTasks

As mentioned above, all of the Tasks found in AIPS are available to ParselTongue. To make use of them, you must first create an AIPSTask object with the name of the AIPS task as the sole argument. For example, you can create an instance of the **MANDL** task with the following:

```
>>> mandl = AIPSTask('MANDL')
```

The variable `mandl` is the handle by which you manipulate this instance of the task. So, to set the AIPS adverbs which govern this task, you only need to set the appropriate attributes on `mandl`:

```
>>> mandl.imsize = AIPSList([256]) # set the image size in cells
>>> mandl.outname = "MBROT"        # set the name of the output image
>>> mandl.outseq = 3               # set output image sequence number
```

To start a task running, simply invoke the `go` method for the task object:

```
>>> mandl.go()
MANDL1: Task MANDL  (release of 31DEC07) begins
MANDL1: Create MBROT       .MANDL .   3 (MA)  on disk  1  cno    1
MANDL1: Appears to have ended successfully
MANDL1: jop56        31DEC07 TST: Cpu=       0.0  Real=       1
>>>
```

To get the full AIPS help file for a task, simply supply an AIPSTask object as an argument to the `help` function:

```
>>> fitld = AIPSTask('FITLD')
>>> help(fitld)
FITLD
Type: Task
Use:  FITLD loads both maps and UV data from tape (or disc) to disc. It
      will only load FITS files, if any other type of file exist on the
      tape the task will fail, so users must skip over non-FITS files.
      The aim of this task is to read, in one pass, a tape written by
                          .  .  .  .  .
```

It is also important to reemphasise that attributes are private to the object to which they belong, so a different instance of the same AIPS task can have completely different attributes:

```
>>> fitld           = AIPSTask('FITLD')
>>> fitldtwo        = AIPSTask('FITLD')
>>> fitld.infile    = "one.fits"
```

```
>>> fitld.outname    = "FIRSTFITS"
>>> fitldtwo.infile  = "two.fits"
>>> fitldtwo.outname = "SECONDFITS"
>>> fitld.go()
FITLD1: Task FITLD  (release of 31DEC07) begins
FITLD1:     FITLD Version 10.5.1
.  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
>>> fitldtwo.go()
FITLD1: Task FITLD  (release of 31DEC07) begins
FITLD1:     FITLD Version 10.5.1
.  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
>>>
```

Array attributes

In the above example, you'll notice that we initialised the `imsize` attribute of `mandl` with a rather unusual contruction. As AIPS descends from FORTRAN, it assumes arrays that are indexed beginning with 1, instead of zero, as is common to modern programming languages. A way around this mismatch is to simply leave the first element of every array blank (or "None", in Python parlance), and begin with the second index. This means that, whenever assigning an array to an attribute of an AIPSTask, an extra null element must be prepended to the array. So, [256] becomes [None,256]. To simplify this bit of tedium, the `AIPSList` function takes a list as its single argument, and returns a list with this null element added.

```
>>> mandl.imsize = [None,256]
>>> mandl.imsize = AIPSList([256]) # identical to the first line
>>> mandl.imsize[1:] = [256] # still more of the same
```

Note that assigning to the first element of such a list will raise a ValueError exception.

Special attributes

There are a number of adverbs which ParselTongue treats in a special fashion, either differently from the interpretation in AIPS, or adverbs which are simply not found in AIPS at all.

`infile`, `outfile`, `outprint`, `infile2`, `ofmfile`, `boxfile`, `oboxfile`

While it possible to specify files by their complete path names since the 31DEC02 version of AIPS there still is a limit on the length of the path name. And 48 characters does not leave you with a lot of leg room. To overcome this limitation, ParselTongue allows you to enter path names of arbitrary length, as long as the trailing component (the filename itself) does not require more than 46 characters. This works with all versions of AIPS, even with versions older than 31DEC02.

```
>>> fitld = AIPSTask('FITLD')
>>> fitld.infile = "/this/sort/of/ridiculously/long/path/would/really/
```

Tip

If you don't specify an area or directory for the `outfile` and `outprint` attributes, AIPS will place the output file in seemingly random locations. If you want these files to end up in the directory from where you run your script, you can prepend ./ to the filename.

`indata`, `in2data`, `in3data`, `in4data`, `outdata`, `out2data`

Having to specify all four of name, class, disk and sequence number to refer to a work file can be a bit of a pain. To make life a bit easier, ParselTongue provides these data attributes. If you have a ParselTongue AIPS data object (see the section AIPSData), you can use it to set these attributes, which set name, class, disk and sequence number in one go.

```
>>> name = "N1066"
>>> class = "UVDATA"
>>> seq = 1
>>> disk = 1
>>> datafile = AIPSUVData(name,class,disk,seq) # construct the AIPSUVDa
>>> calib = AIPSTask('CALIB')
>>> calib.indata = datafile # ...and use it to initialise the AIPSTask
```

`msgkill`

AIPS can be quite chatty, and in long running scripts it may be desirable that only messages about serious problems are displayed. AIPS itself provides the **MSGKILL** pseudoverb and ParselTongue gives you the opportunity to set it, either globally or on a per-task basis, as an attribute of the `AIPSTask`. Unfortunately, setting **MSGKILL** has the drawback that suppressed messages are forever lost and AIPS discourages you from using it. However, in ParselTongue you can set the msgkill attribute to a negative value. This will prevent messages from being displayed on the terminal, as if you set **MSGKILL** to the absolute value of the attribute, but the suppressed messages will still go to the AIPS message file. This way you can always inspect them later if the need arises.

```
>>> AIPSTask.msgkill = -1 # sets global default value for all AIPSTasks
>>> fitld = AIPSTask('FITLD')
>>> fitld.msgkill = 1 # sets the msgkill value for this task only
```

AIPSData

AIPS knows about two rather different types of data, those being images and UV data sets. Once you know the name, class, disk number and sequence number of your data set from the AIPS catalogue, you can create an `AIPSData` object from this information, and use the object as a handle to refer to your data set. For UV data, you'd use the following:

```
>>> uvdata = AIPSUVData('TARGET','UVDATA',1,2)
```

You can probably guess at the analogue for image data:

```
>>> image = AIPSImage('MANDELBROT','MANDL',3,2)
```

In addition to using these objects to initialise `AIPSTask` attributes as seen above, you can access and manipulate your data set directly using `AIPSData` methods. To verify that your data object corresponds to an actual data set in the AIPS catalogue, use the `exists` method:

```
>>> image.exists()
```

```
False
>>> uvdata.exists()
True
```

Manipulation is pretty much limited to deleting extension tables from a data set and deleting the data itself. The latter is a matter of calling the method `zap()` while the former has a somewhat more complicated interface:

`zap_table(`*`type`*`,`*`version`*`)` here type is the type of the extension table as a string (e.g. 'CL') and version is the version to be deleted. Using 0 as the version deletes the highest version of the given type, wheras using -1 deletes them all. So the following example

```
>>> uvdata = AIPSUVData('N04C2', 'UVDATA', 1, 1)
>>> uvdata.zap_table('CL', 0)
```

deletes the last calibration tables from the UV data set N04C2.UVDATA.1 on disk 1. Note that plot files ('PL') and slice files ('SL') can also be deleted. So the following lines

```
>>> imgdata = AIPSImage('MANDELBROT', 'MANDL', 1, 1)
>>> imgdata.zap_table('PL', -1)
```

can be used to throw away all previously made plots.

Deleting a complete data set or image is much simpler. After

```
>>> uvdata.zap()
>>> imgdata.zap()
```

the data and image are forever lost.

Sometimes an AIPS task will fall over and leave your data in a "busy" state. To recover from this you can use

```
>>> uvdata.clrstat()
>>> imgdata.clrstat()
```

but do not do this on data that is being actively used by AIPS or ParselTongue. That would be bad.

Headers

You can access the header through the header attribute:

```
>>> uvdata.header
...
```

```
>>> uvdata.header.telescop
'EVN     '
>>> image.header
...
>>> image.header.date_obs
'2005-03-01'
```

Keywords are the same as for the AIPS GETHEAD verb, but lower case with '-' replaced by '_'.

Tables

What tables do we have?

```
>>> uvdata.tables
[[1, 'AIPS HI'], [1, 'AIPS AT'], [1, 'AIPS NX'], [1, 'AIPS CL'], [1, 'AIPS FQ'], [
```

These tables can be opened for further analysis:

```
>>> sutable = uvdata.table('SU', 1)
>>> for row in sutable:
...     print row.source, row.raapp, row.decapp
...
3C84             49.9489485008 41.513581044
DA193            88.8813073705 39.8151637728
```

It is also possible to access individual rows directly:

```
>>> cltable = uvdata.table('CL', 1)
>>> print cltable[0]
...
```

Convenience functions

ParselTongue includes some convenience functions you may find useful for exploring data headers, which are fairly self-explanatory:

```
>>> uvdata.antennas
['MC', 'WB', 'NT', 'JB']
>>> uvdata.sources
['3C84', 'DA193']
>>> uvdata.polarizations
['R', 'L']
>>> uvdata.stokes
['RR', 'LL', 'RL', 'LR']
>>> image.stokes
['I']
```

Advanced ParselTongue

Manipulating data and tables directly

ParselTongue includes a seperate module for manipulating data beyond simple inspection of metadata. It allows full access to AIPS tables, and access to visibilities and image pixels.

The Wizardry functionality has certain limitations. The most important limitation is that access to remote data is not possible; *only data on the locally visible AIPS disks can be accessed*. This restriction comes about because the XML-RPC protocol used by ParselTongue for remote execution and data access is not suitable for transferring large amounts of data.

Table access in the Wizardry module is similar to normal table access, but with extra functionality for changing existing tables and creating new ones.

New tables can be appended to an `AIPSData` object via the `attach_table` method. The following bit of code creates a new CL table with a different amplitude calibration:

```
from AIPS import AIPS
from Wizardry.AIPSData import AIPSUVData

AIPS.userno = 667

data = AIPSUVData('N03L1', 'UVDATA', 1, 1)
oldcl = data.table('CL', 1)

newcl = data.attach_table('CL', 2, no_term=oldcl.keywords['NO_TERM'])
newcl.keywords['NO_ANT'] = oldcl.keywords['NO_ANT']

for row in oldcl:
row.real1 = [2 * x for x in row.real1]
row.real2 = [2 * x for x in row.real2]
newcl.append(row)

newcl.close()
oldcl.close()
```

As you can see, the table attribute of `AIPSData` objects are exposed as a set of rows, which can be modified as you like:

```
from Wizardry.AIPSData import AIPSUVData as WAIPSUVData
from AIPS import AIPS

AIPS.userno = 667

data = WAIPSUVData('N03L1', 'UVDATA', 1, 1)
table = data.table('CL', 1)

for row in table:
    row.real1 = [2 * x for x in row.real1]
    row.real2 = [2 * x for x in row.real2]
```

```
    row.update()

table.close()
```

Note the `import AIPSUVData as WAIPSUVData` above. While not required, you may find it advantageous to import Wizardry objects with a slightly different name to more easily use both the Wizardry and mundane versions of `AIPSData` objects.

Moving data from one AIPS installation to another

It may be the case that you have a data set within AIPS that needs to be moved to another AIPS installation. Beginning with version 1.1, the ParselTongue distribution includes the **ParselFileServer** program, which runs on the system you wish to transfer files to. Once it is running, you can have your scripts load the Utilities module, and copy data over with the `rcopy` method. Note also the use of the `rdiskappend` method for easily adding proxy disks for use by ParselTongue (the method returns the index in the AIPS.disks array corresponding to the newly-added disk).

```
import Utilities
import AIPS
import AIPSData
import AIPSTask

AIPS.userno = 666
proxy = "http://my.pt.proxy.nl:8000"
rdisk = 3  # The number of the AIPS disk on the remote host

remoteno = Utilities.rdiskappend(proxy,rdisk)

# Set up the local data object, which exists, and the remote
# object, which doesn't exist (until the copy is performed)
ldata = AIPSData.AIPSImage('MANDL','MANDL',1,1)
rdata = AIPSData.AIPSImage('MANDL','MANDL',remoteno,1)

# Perform the copy
Utilities.rcopy(ldata,rdata)
```

Warning

The `rcopy` method uses the /tmp directory on the destination machine as a temporary holding area. If you plan on transferring large files this way, ensure that sufficient space exists on the disk partition in which /tmp lives.

Using external tasks with ParselTongue

A number of useful tasks have been written for AIPS, but are not included in the installation proper. These external tasks can be seen and run by ParselTongue with very little extra trouble.

To get started, let's imagine that we have a task **BLAH.EXE**, which we would like to invoke in a ParselTongue script. We should also have the help file BLAH.HLP, used to generate the task parameter list. Now, to create a directory structure which will contain the external tasks you want to run. You'll need a HELP subdirectory, along with an `$ARCH`/LOAD directory, where `$ARCH` is the name AIPS uses for

the architecture of your machine. This is almost certainly one of LINUX (Intel-based Linux), MACINT (Intel-based Mac OSX), or SOL (Solaris). So, on a PC running Linux, we'd do the following:

```
$ mkdir -p external/HELP
$ mkdir -p external/LINUX/LOAD
$ mv BLAH.HLP external/HELP
$ mv BLAH.EXE external/LINUX/LOAD
```

Next, set an environmental variable for the path of your directory:

```
$ export MYTASKS=/home/sipior/external
```

Fire up ParselTongue as usual, and use the name of the environment variable when creating your `AIPSTask` instance:

```
>>> blah = AIPSTask('BLAH', version='MYTASKS')
>>> blah.adverbs = "whatever"
>>> blah.go()
```

Tip

If you have problems with ParselTongue not seeing a correct list of adverbs, or displaying older versions of help files, remove the appropriate .pickle files from the directory `$HOME`/.ParselTongue, and restart.

Remote ParselTongue execution

ParselTongue can also execute jobs on remote computers, allowing you to make use of powerful, well-equipped machines from the comfort of your laptop, cell phone or whatever.

Executing jobs remotely with ParselTongue is pretty straightforward. ParselTongue uses the XML-RPC protocol to communicate with a ParselTongueServer instance running on the remote machine. So, the first step is to make sure that ParselTongue is installed on both the local and remote machines. Once this is done, start the XML-RPC server on the remote computer:

```
localhost$ ssh remotehost
remotehost% nohup ParselTongueServer &
```

The **nohup** command ensures that ParselTongueServer will not be terminated when you log off the system. Don't forget to ensure that the appropriate LOGIN.SH or LOGIN.CSH file has been sourced before starting the Server.

Once this is done, the following code snippet will set up an `AIPSTask` that will execute on the remote host when its `go()` method is invoked:

```
# Append remote server to global proxies variable. Specify the hostname of
# the remote server, along with the port number on which ParselTongueServer
```

```
# is listening (8000, by default)

AIPS.proxies.append("http://jop01.nfra.nl:8000")

# Instantiate a disk instance for the remote machine, in this case rdisk
# will refer to AIPS disk 1 on the proxy referred to by AIPS.proxies[1]

rdisk = AIPSDisk(AIPS.proxies[1],1)

# Append disk instance to global disks variable

AIPS.disks.append(rdisk)

# Use the appropriate AIPS.disks entry in constructing the data object Here,
# we could replace "2" with "len(AIPS.disks) - 1", since we know that this
# would refer to the entry we just appended to the AIPS.disks list.

data = AIPSImage('MANDL','MANDL',2,1,666)

# provide appropriate data objects for task parameters

job = AIPSTask('mandl')
job.outdata = data
# And we're off, on the proxy listening on port 8000 of host jop01.nfra.nl
job.go()
```

Parallel ParselTongue execution

As of version 1.1, ParselTongue is capable of running multiple AIPS tasks simultaneously. To be clear, this is not true parallelisation, with active communication between the computational nodes. Fundamentally, ParselTongue despatches AIPS tasks, and so a truly parallel ParselTongue would require a parallel AIPS. However, many data reduction problems benefit greatly from so-called "trivial parallelisation", which is a fancy way of saying that if you have four independent data sets, and four computers, you can handily save yourself a great deal of time by running four data reduction processes simultaneously.

With that proviso, let's talk about how one gets ParselTongue to do several things at once. The `ParallelTask` module contains all you need to arrange this. It contains the `ParallelQueue` and `ParallelTask` classes, but you will only need to make use of the former (which is simply a container for `ParallelTask` objects) directly. You can create your `AIPSTask` objects just as before, and add them to your work queue with the `ParallelQueue.queue` method. Once you have added a set of tasks, the `ParallelQueue.go` method will despatch them all to the various hosts pointed to by the relevant `AIPSData` objects.

In addition, the logging facility typically employed in a ParselTongue run (where messages are dumped to the terminal, and/or a single file), has some problems when dealing with multiple tasks, all sending back messages to the user in an essentially random order. To establish some useful order over your output, consider setting the `log` attribute for each AIPSTask in your task queue. As discussed in the Logging section, setting this attribute to a file object causes all AIPS output from that task to be redirected there.

An example might be useful here. The following script executes two simple processes (the **MANDL** task from AIPS), on two separate machines (jop01 and jop02). Both of the remote hosts are, of course, running an instance of the **ParselTongueServer** daemon to handle the remote task execution.

```
import ParallelTask
import time, sys
import Utilities, AIPS, AIPSData, AIPSTask

AIPS.userno = 666

queue = ParallelTask.ParallelQueue()

# Add the desired ParselTongue proxies, and entries for the remote disks
# that will hold the task output.
AIPS.proxies.append("http://jop01.nfra.nl:8000")
AIPS.proxies.append("http://jop02.nfra.nl:8000")
rdisk1 = AIPS.AIPSDisk(AIPS.proxies[len(AIPS.proxies)-1],1)
rdisk2 = AIPS.AIPSDisk(AIPS.proxies[len(AIPS.proxies)-1],2)
AIPS.disks.append(rdisk1)
AIPS.disks.append(rdisk2)

data1 = AIPSData.AIPSImage('MANDL','MANDL',len(AIPS.disks)-2,1,AIPS.userno)
data2 = AIPSData.AIPSImage('MANDL','MANDL',len(AIPS.disks)-1,2,AIPS.userno)

mandl1 = AIPSTask.AIPSTask('MANDL')
mandl2 = AIPSTask.AIPSTask('MANDL')
mandl1.outdata = data1
mandl2.outdata = data2
mandl1.imsize = [None,4096,4096]
mandl2.imsize = [None,4096,4096]

# Set the task output logs
mandl1.log = open("mandl1.log","a")
mandl2.log = open("mandl2.log","a")

# Add these tasks to the work queue
queue.queue(mandl1)
queue.queue(mandl2)

# And we're off!
queue.go()
```

Warning

If you plan on running ParselTongue on a cluster with shared disk place, ensure that the first AIPS disk is unique for every node. That is, the first entry in the file ${AIPS_ROOT}/DA00/DADEVS.LIST must be unique for each host. If this first disk is shared among many nodes, the message log produced by AIPS will be a jumbled mess after running a job queue. Unfortunately, the location of the message file does not appear to be configurable under AIPS, making this restriction fundamental.

Logging

ParselTongue provides a simple, yet powerful way to capture a log of your session. If you set `AIPS.log` to a file object all output from AIPS tasks will be sent to that object:

```
>>> AIPS.log = open('ParselTongue.log', 'a')
```

In fact, the above is not the full truth. `AIPS.log` does not necessarily have to be a true Python file object; any object with a write method will do.

The output sent to `AIPS.log` includes the messages that are not being displayed on your terminal because the `msgkill` attribute was set to a negative value in the task object (this does not apply to messages killed by a positive value of this attribute; those are lost forever).

To stop logging, set `AIPS.log` to None.

Per-task logging

In a number of cases, it may be helpful for each AIPS task to save its output messages in a separate log file. This is especially true when employing parallel job queues, detailed in the Parallel ParselTongue execution section above. To get a separate message log for your task, simply set the `log` attribute to a file object:

```
>>> mandl = AIPSTask('MANDL')
# This sets the log for the mandl AIPSTask to the file mandl1.log
>>> mandl.log = open('mandl1.log','a')
```

ParselTongue reference

A complete index of the Classes that comprise ParselTongue, along with their methods and docstrings can be found here [http://www.jive.nl/parseltongue/doc/index.html].