1.7.1 table - Tool

Package utility
Module table

Access tables from Glish

Description

table is the tool that contains all the tool functions relevant for table handling.
To make a table tool, use the constructor.

Constructors

table	Construct table tool
tablefromascii	Create an AIPS++ table from an ASCII format
tablefromfits	Create an AIPS++ table from binary FITS

Functions

addcols	add one or more columns
addreadmeline	add a readme line to the info record
addrows	add a specified number of rows
browse	browse a table using a graphical browser
calc	Calculate an expression on a table
close	close the table tool
colarraytype	return the column array type
coldatatype	return the column data type
colfieldnames	get the names of fields in a keyword in a column
colkeywordnames	get the names of all keywords in a column
colnames	return the names of the columns
copy	copy a table
copyrows	copy rows from this table to another
datachanged	has data changed in table?
done	end the table tool
endianformat	get the endian format used for this table
fieldnames	get the names of fields in a table keyword
flush	flush the current contents to disk
getcell	get a specific cell
getcellslice	get a slice from a specific cell
getcol	get a specific column
getcoldesc	get the description of a specific column
getcolkeyword	get value of specific column keyword
getcolkeywords	get values of all keywords for a column
getcolshapestring	get shape of arrays in a specific column
getcolslice	get a slice from a specific column

getdesc	get the table description
getdminfo	get the info about data managers
getkeyword	get value of specific table keyword
getkeywords	get values of all table keywords
getvarcol	get a specific column (for variable arrays)
haslock	has this process a lock on the table?
info	get the info record
iscelldefined	test if a specific cell contains a value
ismultiused	is the table in use in another process?
isscalarcol	is the specified column scalar?
isvarcol	tell if column contains variable shaped arrays
iswritable	is the table writable?
keywordnames	get the names of all table keywords
lock	acquire a lock on the table
lockoptions	get the lock options used for this table
name	return name on disk
ncols	return number of columns
nrows	return number of rows
ok	Is the table tool ok?
putcell	put a specific cell
putcellslice	put a slice into a specific cell
putcol	put a specific column
putcolkeyword	put a specific keyword for a column
putcolkeywords	put multiple keywords for a column
putcolslice	put a slice into a specific column
putinfo	set the info record
putkeyword	put a specific table keyword
putkeywords	put multiple table keywords
putvarcol	put a specific column (for variable arrays)
query	Make a table from a query
removecolkeyword	remove a specific keyword for a column
removecols	remove one or more columns
removekeyword	remove a specific table keyword
removerows	remove the specified rows
renamecol	rename a column
resync	resync the table tool with table file
rownumbers	return the row numbers in the (reference) table
selectrows	Make a table from a selection of rows
setmaxcachesize	set maximum cache size for column in the table
summary	summarize the contents of the table
toascii	Write AIPS++ table into an ASCII format
unlock	unlock and flush the table

table - Constructor

Package **utility**

Module	**table**
Tool	**table**

Construct table tool

Synopsis
table(tablename, tabledesc, nrow, readonly, lockoptions, ack, dminfo, endian, memorytable, tableserver, tablenote, handle)

Description
Use this constructor to construct a table tool inside Glish from the name of a disk file containing an AIPS++ Table. A new table may also be created from a table descriptor (see tablecreatedesc). When creating a new table, detailed data manager information can be given using the `dminfo` argument. This is a record as returned by the getdminfo function.
Most of the arguments are rarely used: most of the time, you'll just need to use the tablename, and perhaps readonly.
A table can be shared by multiple processes by using the appropriate locking options. The possible options are:
- auto: let the system take care of locking. At regular time intervals these autolocks are released to give other processes the opportunity to access the table. The aipsrc variable
`table.relinquish.reqautolocks.interval` defines the number of seconds between releasing autolocks on tables needed in another process.
`table.relinquish.allautolocks.interval` defines the number of seconds between releasing all autolocks.
- autonoread: as auto, but no read locking is needed. This must be used with care, because it means that reading can be done while the table tool is not synchronized with the table file (as is normally done when a lock is acquired). The function `resync` can be used to explicitly synchronize the table tool
- user: the user takes care by explicit calls to lock and unlock
- usernoread: as user and the no readlocking behaviour of autonoread.
- permanent: use a permanent lock; the constructor fails when the table is already in use in another process
- permanentwait: as above, but wait until the other process releases its lock
- default: this is the default option. If the given table is already open, the locking option in use is not changed. Otherwise it reverts to auto.
When auto locking is used, it is possible to give a record containing the fields option, interval, and/or maxwait. In this way advanced users have full control over the locking options. In practice this is hardly ever needed.
When creating a new table, the endian format in which the data should be stored, can be specified. The possible values are:
- big: big endian format (as used on e.g. SUN)

- little: little endian format (as used on e.g. PC)
- local: use the endian format of the machine being used
- aipsrc: use the endian format specified in aipsrc variable table.endianformat (which defaults to big).

The default is aipsrc.
Note that usually it is best to store data in local endian format, because that requires the least amount of byte swapping. However, if the table must be accessible with AIPS++ version 1.7 or before, big endian should be used.

When creating a new table, the table will normally reside on disk. It is, however, possible to specify that the table should be held in memory. In such a case the table is process specific, thus cannot be seen by other processes. Note that a memory table uses the MemoryStMan storage manager for all its stored columns, but it is still possible to use virtual columns as well.

Arguments

tablename	in	Name of table on disk Allowed:String—Bool Default: F
tabledesc	in	Table descriptor Allowed:Descriptor—Bool Default: F
nrow	in	Number of rows Allowed:Int Default: 0
readonly	in	Open Read-only? Allowed:Bool Default: T
lockoptions	in	locking to be used Allowed:String—Record Default: default
ack	in	Acknowledge creations, etc Allowed:Bool Default: T
dminfo	in	Data manager info for new table Allowed:Record Default: F
endian	in	Endian format of new table Allowed:String Default: aipsrc
memorytable	in	Hold new table in memory? Allowed:Bool Default: F
tableserver	in	Table handler to be used Allowed:Any tableserver Default: defaulttableserver
tablenote	in	note function to be used Allowed:Any note function Default: note
handle	in	attach to this table handle Allowed:for internal use only Default: F

Returns
table tool

Example

```
table1:=table("3C273XC1.MS");
table1.browse();
```

The first line opens an existing table 3C273XC1.MS, the second browses it using the browse tool function.

```
table2:=table("name", readonly=F, lockoptions='user', endian='local');
table2.lock();
table2.addRows();
table2.unlock();
```

In this example explicit user locking is used. The function lock is needed to acquire a (write) lock before the addRows is done. Thereafter the lock is released to give other processes the chance to operate on the table.
Note that releasing a lock implies flushing the table, so doing that very often can be quite expensive.

---

tablefromfits - Constructor

Package	**utility**
Module	**table**
Tool	**table**

Create an AIPS++ table from binary FITS

Synopsis
tablefromfits(tablename, fitsfile, whichhdu, storage, convention, readonly, lockoptions, ack, tableserver)

Description
Create a table from binary FITS format. This generates an AIPS table from the binary FITS table in the given HDU (header unit) of the FITS file. Note that other FITS formats (*e.g.* Image FITS and UVFITS) are read by other means.
It is possible to specify the storage manager to use for the table:
`standard` is the default storage manager.
`incremental` is efficient for slowly varying data.

Arguments

tablename	in	Name of table to be created Allowed:String
fitsfile	in	Name of FITS file to be read Allowed:String
whichhdu	in	Which HDU to read (0-relative to primary HDU i.e. 1 is the smallest valid value) Allowed:Int Default: 1
storage	in	Storage manager to use (standard or incremental) Allowed:String Default: standard
convention	in	Convention to use (SDFITS or none) Allowed:String Default: sdfits
readonly	in	Open Read-only? Allowed:Bool Default: T
lockoptions	in	locking to be used Allowed:String—Record Default: default
ack	in	Acknowledge creations, etc Allowed:Bool Default: T
tableserver	in	Table server to be used Allowed:Any tableserver Default: defaulttableserver

Returns
table tool

---

tablefromascii - Constructor

Package	**utility**
Module	**table**
Tool	**table**

Create an AIPS++ table from an ASCII format

Synopsis
tablefromascii(tablename, asciifile, headerfile, autoheader, autoshape, sep, commentmarker, firstline, lastline, readonly, lockoptions, ack, tableserver)

Description
Create a table from a file in ASCII format. Columnar data as well as table and column keywords may be specified.
Once the table is created from the ASCII data, it is opened in the specified mode and a table tool is returned.
The table columns are filled from a file containing the data values separated by a separator (one line per table row). The default separator is a blank. Blanks after the separator are ignored.
If a non-blank separator is used, values can be empty. Such values default to 0, empty string, or F depending on the data type. E.g. 1,,2, has 4 values of which the 2nd and 4th are empty and default to 0. Similarly if fewer values are given than needed, the missing values get the default value.
Either the data format can be explicitly specified or it can be found automatically. The former gives more control in ambiguous situations. Both scalar and array columns can be generated from the ASCII input. The format string determines the type and optional shape.
In automatic mode (`autoheader=T`) the first line of the ASCII data is analyzed to deduce the data types. Only the types I, D, and A can be recognized. A number without decimal point or exponent is I (integer), otherwise it is D (double). Any other string is A (string). Note that a number may contain a leading sign (+ or -). The `autoshape` argument can be used to specify if the input should be stored as multiple scalars (the default) or as a single array. In the latter case one axis in the shape can be defined as variable length by giving it the value 0. It means that the actual array shape in a row is determined by the number of values in the corresponding input line.
Columns get the names `Column1`, `Column2`, etc..
For example:

1. `autoshape=[]` (which is the default) means that all values are to be stored as scalar columns.

2. `autoshape=0` means that all values in a row are to be stored as a variable length vector.

3. `autoshape=10` defines a fixed length vector. If an input line contains less than 10 values, the vector is filled with default values. If more than 10 values, the latter values are ignored.

4. `autoshape=[5,0` defines a 2-dim array of which the 2nd axis is variable. Note that if an input line does not contain a multiple of 5 values, the array is filled with default values.

If the format of the table is explicitly specified, it has to be done either in the first two lines of the data file (named by the argument filename), or in a separate header file (named by the argument headerfile). In both forms, table keywords may also be specified before the column definitions. The column names and types can be described by two lines:

1. The first line contains the names of the columns. These names may be enclosed in quotes (either single or double).

2. The second line contains the data type and optionally the shape of each column. Valid types are:

   - S for Short data
   - I for Integer data
   - R for Real data
   - D for Double Precision data
   - X for Complex data (Real followed by Imaginary)
   - Z for Complex data (Amplitude then Phase)
   - DX for Double Precision Complex data (Real followed by Imaginary)
   - DZ for Double Precision Complex data (Amplitude then Phase)
   - A for ASCII data (a value must be enclosed in single or double quotes if it contains whitespace)
   - B for Boolean data (False are empty string, 0, or any string starting with F, f, N, or n).

If a column is an array, the shape has to be given after the data type without any whitespace. E.g. `I10` defines an integer vector of length 10. `A2,5` defines a 2-dim string array with shape [2,5]. Note that `I` is not the same as `I1` as the first one defines a scalar and the other one a vector with length 1. The last column can have one variable length axis denoted by the value 0. It "consumes" the remainder of the input line.
If the argument headerfile is set then the header information is read from that file instead of the first lines of the data file.
To give a simple example of the form where the header information is located at the top of the data file:

```
COLI   COLF   COLD       COLX        COLZ       COLS
 I      R      D          X           Z          A
1      1.1    1.11       1.12 1.13   1.14 1.15  Str1
10     11     12         13   14     15   16    ""
```

Note that a complex number consists of 2 numbers.
Also note that an empty string can be given.
Let us now give an example of a separate header file that one might use to get interferometer data into AIPS++:

```
U     V      W         TIME        ANT1       ANT2      DATA
R     R      R          D           I          I        X1,0
```

The data file would then look like:

```
124.011 54560.0  3477.1  43456789.0990    1      2        4.327 -0.1132
34561.0 45629.3  3900.5  43456789.0990    1      3        5.398 0.4521
```

Note that the DATA column is defined as a 2-dim array of 1 correlation and a variable number of channels, so the actual number of channels is determined by the input. In this example both rows will have 1 channel (note that a complex value contains 2 values).
Tables may have keywords in addition to the columns. The keywords are useful for holding information that is global to the entire table (such as author, revision, history, *etc,*).
The keywords in the header definitions must preceed the column descriptions. They must be enclosed between a line that starts with ".key..." and a line that starts with ".endkey..." (where ... can be anything). Between these two lines each line should contain the following as listed below. A table keywordset and column keywordsets can be specified. The latter can be specified by specifying the column name after the .keywords string.

- The keyword name, e.g., ANYKEY
- The datatype and optional shape of the keyword (cf. list of valid types above)
- The value or values for the keyword (the keyword may contain a scalar or an array of values). e.g., 3.14159 21.78945

Thus to continue the example above, one might wish to add keywords as follows:

```
.keywords
DATE        A  "97/1/16"
REVISION    D 2.01
AUTHOR      A "Tim Cornwell"
INSTRUMENT  A "VLA"
.endkeywords
.keywords TIME
UNIT A "s"
.endkeywords
U     V      W         TIME        ANT1       ANT2      DATA
R     R      R          D           I          I        X1,0
```

Similarly to the column format string, the keyword formats can also contain shape information. The only difference is that if no shape is given, a keyword can have multiple values (making it a vector).
It is possible to ignore comment lines in the header and data file by giving the `commentmarker`. It indicates that lines starting with the given marker are ignored. Note that the marker can be a regular expression (e.g. texttt' *//' tells that lines starting with // and optionally preceeded by blanks have to be ignored).

With the arguments `firstline` and `lastline` one can specify which lines have to be taken from the input file. A negative value means 1 for `firstline` or end-of-file for `lastline`. Note that if the headers and data are combined in one file, these line arguments apply to the whole file. If headers and data are in separate files, these line arguments apply to the data file only.

Also note that ignored comment lines are counted, thus are used to determine which lines are in the line range.

The number of rows is determined by the number of lines read from the data file.

Arguments

tablename	in	Name of table to be created Allowed:String
asciifile	in	Name of ASCII file to be read Allowed:String
headerfile	in	Name of an optional file defining the format Allowed:String Default: ”
autoheader	in	Determine header information automatically Allowed:Boolean Default: F
autoshape	in	Shape to be used if autoheader=T Allowed:Vector of Ints Default: []
sep	in	Value separator Allowed:String Default: ' '
commentmarker	in	Regex indicating comment line Allowed:String Default: ”
firstline	in	First line to use Allowed:Int Default: 1
lastline	in	Last line to use Allowed:Int Default: -1
readonly	in	Open Read-only? Allowed:Bool Default: T
lockoptions	in	locking to be used Allowed:String—Record Default: default
ack	in	Acknowledge creations, etc Allowed:Bool Default: T
tableserver	in	Table server to be used Allowed:Any tableserver Default: defaulttableserver

Returns
table tool

---

table.flush - Function

Package **utility**
Module **table**

Tool table

flush the current contents to disk

Synopsis
flush(recursive)

Description
Until a flush is performed, the results of all operations are not reflected in any change to the disk file. Hence you *must* do a flush to write the changes to disk.

Arguments

recursive	in	Flush all subtables recursively? Allowed:Bool Default: F

Returns
Bool

---

table.resync - Function

Package	**utility**
Module	**table**
Tool	**table**

resync the table tool with table file

Synopsis
resync()

Description
Acquiring a read or write lock automatically synchronizes the internals of the table tool with the actual contents of the table files. In this way different processes accessing the same table always use the same table data.
However, a table can be used without read locking. In that case the table tool internals are not synchronized automatically. The resync function offers a way

to do explicit synchronization. It is only useful if the table is opened with locking mode `autonoread` or `usernoread`.

Returns
Bool

---

table.close - Function

Package	**utility**
Module	**table**
Tool	**table**

close the table tool

Synopsis
close()

Description
First a flush is done, then the table is closed inside Glish and is no longer available for use.

Returns
Bool

---

table.copy - Function

Package	**utility**
Module	**table**
Tool	**table**

copy a table

Synopsis
copy(newtablename, deep, valuecopy, dminfo, endian, memorytable, returnobject, copynorows)

Description

Copy the table. All subtables are also copied. References to another table are preserved.

The argument `deep` determines how a reference table (i.e. the result of a query) is copied. By default a file copy is made, thus the resulting table still contains references and no actual data. If, however, `deep=T` is given, a deep copy is made which means that the actual data are copied. Also all subtables are copied.
Normally a plain table is copied by copying the files. However, if `deep=T` and `valuecopy=T` are given, a plain table is copied by copying all its values and subtables. This is useful to reorganize the tables, i.e. to regain file space that is wasted by frequent updates to a table.
The argument `dminfo` can be used to specify explicit data manager info for the columns in the new plain table. It can be used to change, for example, a storage manager from IncrStMan to StandardStMan. The `dminfo` is a record as returned by the getdminfo If `dminfo` is a non-empty record, it forces `valuecopy=T`.

By default all data are copied when a table and its subtables are copied. However, by giving `copynorows=T` the rows are not copied, thus only the table structure and keywords are copied.

The standard operation is make the copy to a plain table. It is, however, possible to copy to a memory table by giving `memorytable=T`.

The endian format for the newly created table can be specified. This is only meaningful if a deep copy is made to a plain table. The possible values are:
- big: big endian format (as used on e.g. SUN)
- little: little endian format (as used on e.g. PC)
- local: use the endian format of the machine being used
- aipsrc: use the endian format specified in aipsrc variable table.endianformat (which defaults to big).
The default is aipsrc.

Normally the `copy` function only copies the table and does not create a new table tool object. The user can do that by opening the newly created table in the standard way. However, it is possible to get an object back by using `returnobject=T`. An object is always returned if the copy is made to a memory table.

Arguments

newtablename	in	Name of newtable on disk Allowed:String
deep	in	Make a deep copy of a reference table? Allowed:Bool Default: F
valuecopy	in	Make a deep copy of any table? Allowed:Bool Default: F
dminfo	in	Data manager info for new table Allowed:Record Default: [=]
endian	in	Endian format of new table Allowed:String Default: aipsrc
memorytable	in	Hold new table in memory? Allowed:Bool Default: F
returnobject	in	Return a tool object for the new table Allowed:Bool Default: F
copynorows	in	Make no copy of the rows? Allowed:Bool Default: F

Returns
Bool

---

table.copyrows - Function

Package	**utility**
Module	**table**
Tool	**table**

copy rows from this table to another

Synopsis
copyrows(outtable, startrowin, startrowout, nrow)

Description
Copy rows from this table to another. By default all rows of this table are appended to the output table. It is possible though to control which rows are copied.

Rows are added to the output table as needed. Because no rows can be added to a reference table, it is only possible to overwrite existing rows in such tables. Only the data of columns existing in both tables will be copied. Thus by making a reference table consisting of a few columns, it is possible to copy those columns only.

Arguments

outtable	in	table object of output table Allowed:String
startrowin	in	First row to take from input table Allowed:Int Default: 1
startrowout	in	First row to write in output table Allowed:Int Default: -1 (=end)
nrow	in	Nr of rows to copy Allowed:Int Default: -1 (=all)

Returns
Bool

Example
This example appends rows to the table itself, thus doubles the number of rows.

```
t:=table('test.ms',readonly=F)
t.copyrows(t)
```

This example copies 10 rows of the selected subset of the MS to the beginning of the output MS.

```
t:=table("3C273XC1.MS")
tout:=table('test.ms',readonly=F)
t1:=t.query('ANTENNA1==0')
t1.copyrows(tout,nrow=10,startrowout=1)
```

---

table.done - Function

Package	**utility**
Module	**table**
Tool	**table**

end the table tool

Synopsis
done()

Description
End the table tool and free up all memory. It cannot be reused anymore.

---

table.iswritable - Function

Package	**utility**
Module	**table**
Tool	**table**

is the table writable?

Synopsis
iswritable()

Description
Test if the table is opened for write.
Note that the global function `tableiswritable(tablename)` tests if a table with the given name is writable.

Returns
Bool

---

table.endianformat - Function

Package	**utility**
Module	**table**
Tool	**table**

get the endian format used for this table

Synopsis
endianformat()

Description
Get the endian format used for this table. It returns a string with value 'big' or 'little'.

Returns
String

---

table.lock - Function

Package	**utility**
Module	**table**
Tool	**table**

acquire a lock on the table

Synopsis
lock(write, nattempts)

Description
Try to acquire a read or write lock on the table. Nothing will be done if the table is already correctly locked by this process. It is only needed when user locking is used. When the lock is acquired, the internal caches will be synchronized with the (possible changed) contents of the table.
It is possible to specify the number of attempts to do (1 per second) in case the table is locked by another process. The default 0 is trying indefinitely.

Arguments

write	in	Write lock? (F=read lock) Allowed:Bool Default: T
nattempts	in	Nr of attempts Allowed:Int Default: 0

Returns
Bool

---

table.unlock - Function

Package	**utility**
Module	**table**
Tool	**table**

unlock and flush the table

Synopsis
unlock()

Description
The table is flushed and the lock on the table is released. This function is only needed when user locking is used. However, it is also possible to use it with auto locking. In that case the lock will automatically be re-acquired before the next table operation.

Returns
Bool

---

table.datachanged - Function

Package	**utility**
Module	**table**
Tool	**table**

has data changed in table?

Synopsis
datachanged()

Description
This function tests if data in the table have changed (by another process) since the last call to this function.

Returns
Bool

---

table.haslock - Function

Package	**utility**
Module	**table**
Tool	**table**

has this process a lock on the table?

Synopsis
haslock(write)

Description
Has this process a read or write lock on the table?

Arguments

write	in	Has it a write lock? (F=read lock) Allowed:Bool Default: T

Returns
Bool

---

table.lockoptions - Function

Package	**utility**
Module	**table**
Tool	**table**

get the lock options used for this table

Synopsis
lockoptions()

Description
Get the lock options used for this table. It returns a record with the fields option, interval and maxwait. The record can be used as the lockoptions argument in the table constructor.

Returns
Record

---

table.ismultiused - Function

Package	**utility**
Module	**table**
Tool	**table**

is the table in use in another process?

Synopsis
ismultiused(checksubtables)

Description
Is the table still in use in another process? If so, the table cannot be deleted.

Arguments

checksubtables	in	check if subtables are multiused?) Allowed:Bool Default: F

Returns
Bool

---

table.browse - Function

Package	**utility**
Module	**table**
Tool	**table**

browse a table using a graphical browser

Synopsis
browse()

Description

To start the browser, the environment variable DISPLAY must be set.

Returns
Bool

---

table.name - Function

Package	**utility**
Module	**table**
Tool	**table**

return name on disk

Synopsis
name()

Description
This gives the name of the AIPS++ table on disk that this table tool corresponds to *i.e.* this is the name that the table was constructed from.

Returns
String

Example

```
  t:=table("3C273XC1.MS")
  t.name()
# 3C273XC1.MS
```

---

table.toascii - Function

Package	**utility**
Module	**table**
Tool	**table**

Write AIPS++ table into an ASCII format

Synopsis
toascii(asciifile, headerfile, columns, sep)

Description
Write a table into an ASCII format. Both columnar data (scalar) and table keywords may be written. The separator between values can be specified and defaults to a blank. Note that columns containing arrays will be ignored (after showing a warning).
The output format is that read by the constructor tablefromascii.
If the argument headerfile is set then the header information is written to that file instead of the first two lines of the data file.

Arguments

asciifile	in	Name of ASCII file to be written Allowed:String
headerfile	in	Name of an optional file defining the format Allowed:String Default: ”
columns	in	Names of columns to be written Allowed:Vector of strings Default: All
sep	in	Value separator Allowed:String Default: ’ ’

Returns
Bool

---

table.query - Function

Package	**utility**
Module	**table**
Tool	**table**

Make a table from a query

Synopsis
query(query, name, sortlist, columns)

Description

Make a table from a query applied to the current table. The query is the qualification part of the tablecommand. It is also possible to specify column(s) and/or expressions to sort on and to specify the columns to be contained in the output table. See the example below. A new table tool is returned. The new (reference) table can be given a name and will then be written to disk. Note that the resulting table is just a reference to the original table. One can make a deep copy of the query result using the copy function (see example). If the arguments `query`, `sortlist`, and `columns` are empty strings (their default values), the taqlwidget will be started (if a GUI is available) before the query function is executed. In that way it is possible to let the user form the query interactively.

Arguments

query	in	Query string Allowed:String Default: ”
name	in	Name of resulting reference table Allowed:String Default: ”
sortlist	in	Sort string (one or more expressions separated by commas) Allowed:String Default: ”
columns	in	List of column names separated by commas Allowed:String Default: ”

Returns

table tool

Example

```
  t:=table("3C273XC1.MS");
  subt:=t.query("SPECTRAL_WINDOW_ID==1",
                sortlist="ARRAY_ID", columns="TIME, DATA, UVW");
  print subt.ncols();
# 3
  t.close();
  subt.copy ("3C273XC1_spw1.MS", T);
  subt.close();
```

From the original table corresponding to the disk file 3C273XC1.MS, only rows with SPECTRAL_WINDOW_ID equal to 1 are selected and sorted by ARRAY_ID. Only the columns TIME DATA UVW are written. Thereafter a deep copy of the result is made. The table query command is equivalent to:

```
subt:=tablecommand(paste('select TIME, DATA, UVW from 3C273XC1.MS',
                         'where SPECTRAL_WINDOW_ID==1',
                         'orderby ARRAY_ID'))
```

The sortlist argument can be used to sort in ascending or descending order (or a mix of them) on one or more columns. Default is ascending. It is also possible to remove duplicate values using the word NODUPLICATES at the beginning. E.g.:

```
sortlist='TIME desc'
sortlist='noduplicates ANTENNA1,ANTENNA2'
sortlist='ANTENNA1 desc, ANTENNA2 asc'
sortlist='desc ANTENNA1, ANTENNA2, TIME'
```

See Also
tablecommand

---

table.calc - Function

Package	**utility**
Module	**table**
Tool	**table**

Calculate an expression on a table

Synopsis
calc(expr)

Description
Get the result from the calculation of an expression on a table. using a CALC command given to the tablecommand function. The expression can be any expression that can be given in the WHERE clause of a SELECT expression (thus including subqueries). The given expression determines if the result is a scalar, a vector, or a record containing arrays. See the examples below.

Arguments

expr	in	Expression string Allowed:String

Returns
record, vector, or scalar

Example

```
t:=table("3C273XC1.MS");
t.calc ('DATA+1');
```

returns a (potentially enormous) record where a field contains the value of the expression for the row with that number. Note that it returns a record because for each row the expression results in an array. It should be clear that this example is useless. However, something like this could be useful for a column with (very) small arrays.

```
t:=table("3C273XC1.MS");
t.calc ('ntrue(FLAG)');
```

returns for each row the number of flags set. The result is a vector, because for each row the expression results in a scalar.

```
t:=table("3C273XC1.MS");
t.calc ('sum([select from $1 giving [ntrue(FLAG)]])');
```

returns the total number of flags set in the table (in a single scalar). It is the same as

```
sum(t.calc ('ntrue(FLAG)'));
```

The latter is easier to write, but sends (much) more data over the glish bus.

See Also
tablecommand

---

table.selectrows - Function

Package	**utility**
Module	**table**
Tool	**table**

Make a table from a selection of rows

Synopsis
selectrows(rownrs, name)

Description
Create a (reference) table containing a given subset of rows. It is, for instance, useful when a selection is done on another table containing the row numbers in the main table. It can be useful to apply the Glish function unique to those row numbers, otherwise the same row might be included multiple times (see example).
Note that usually such referencing row numbers are 0-based, while the selectrows function expects 1-based row numbers. It means that in such cases 1 should be added (see example).
It is possible to give a name to the resulting table. If given, the resulting table is made persistent with that table name. Otherwise the table is transient and disappears when closed or when Glish exits.
The rownumbers function returns a vector containing the row number in the main table for each row in the selection table. Thus given a row number vector `rownrs`, the following is always true.

```
rownrs == maintable.selectrows(rownrs).rownumbers()
```

However, it is not true when selectrows is used on a selection table. because `rownumbers` does not return the row number in that selection table but in the main table.
It means that one has to take great care when using `selectrows` on a selection table.

Arguments

rownrs	in	1-based Row Numbers Allowed:Vector of Ints
name	in	Name of resulting table Allowed:Any String Default: ”

Example

```
#   Do the query on the main table.
  maintable := table('SOMENAME');
  scantable := maintable.query(command);
```

```
#   Get the column containing the 0-based row numbers in the BACKEND table.
#   Make the row numbers unique and make them 1-based.
  backrows := unique(scantable.getcol('NS_GBT_BACKEND_ID')) + 1;
#   Form the table subset of the BACKEND table containing those rows.
  backtable := table('SOMENAME/GBT_BACKEND');
  scanback := backtable.selectrows(backrows);
#   Do something with that table.
  print scanback.nrows();
```

The last statement is equivalent to the query command:

```
  scanback := backtable.query("rownumber() in $backrows");
```

However, the query takes much, much, much more time to execute, especially for larger tables and larger row number vectors. Furthermore, the `backrows` variable need to be global to be able to substitute it in the query command.

Returns
table tool

---

table.putinfo - Function

Package	**utility**
Module	**table**
Tool	**table**

set the info record

Synopsis
putinfo(value)

Description
The info record contains information on the table. It is written by applications, and used by, for example, the catalog directory browser to determine what type of information is stored in a table.

Arguments

value	in	Info record Allowed:info record

Returns
Bool

Example

```
  include 'catalog.g'
  dc.whatis("MS")
# [type=Measurement Set, istable=T, subType=observed]
  dc.whatis("tcal")
# [type=Calibration, istable=T, subType=T Jones]
```

---

table.info - Function

Package	**utility**
Module	**table**
Tool	**table**

get the info record

Synopsis
info()

Description
The info record contains information on the table. It is written by applications, and used by, for example, the catalog directory browser to determine what type of information is stored in a table.

Arguments

Returns
Record

---

table.addreadmeline - Function

Package	**utility**
Module	**table**
Tool	**table**

add a readme line to the info record

Synopsis
addreadmeline(value)

Description
A readme line is part of the info record associated with a table. It is to inform the user, and is not used by any application directly.

Arguments

value	in	readme line Allowed:String

Returns
Bool

---

table.summary - Function

Package	**utility**
Module	**table**
Tool	**table**

summarize the contents of the table

Synopsis
summary(recurse)

Description
A (terse) summary of the table contents is sent to the defaultlogger.

Arguments

recurse	in	Summarize subtables recursively Allowed:Bool Default: F

Returns
Bool

Example

```
  table("tcal").summary()
# successful readonly open of table  tcal :  9 columns, 11 rows
# Table summary: tcal
# Shape: 9 columns by 11 rows
# Info: [type=Calibration, subType=T Jones, readme=]
# Table keywords: [Type=T Jones, Interval=30, DeltaT=1]
# Columns: StartTime StopTime Gain SolutionOK Fit FitWeight
# iSolutionOK iFit iFitWeight
```

---

table.colnames - Function

Package	**utility**
Module	**table**
Tool	**table**

return the names of the columns

Synopsis
colnames()

Description
The names of the columns in the table are returned as a vector of Strings.

Returns
Vector of Strings

Example

```
  table("tcal").colnames()
# StartTime StopTime Gain SolutionOK Fit FitWeight iSolutionOK iFit iFitWeight
```

---

table.rownumbers - Function

Package	**utility**
Module	**table**
Tool	**table**

return the row numbers in the (reference) table

Synopsis
rownumbers(tab, nbytes)

Description
This function can be useful after a selection or a sort. It returns the row numbers of the rows in this table with respect to the given table. If no table is given, the original table is used.
For example:

```
  t:=table('W53.MS')
  t1:=t.selectrows([1,3,5,7,9])
  t1.rownumbers(t)
# [1 3 5 7 9]
  t2:=t1.selectrows([2,5])
  t2.rownumbers(t1)
# [2 5]
  t2.rownumbers(t)
# [3 9]
  t2.rownumbers()
# [3 9]
```

The last statements show that the function returns the row numbers referring to the given table. Table t2 contains rows 2 and 5 in table t1, which are rows 3 and 9 in table t.
Note that when a table is opened using its name, that table can be a reference table. Thus in the example above the last 2 statements may give different results depending on the fact if W53.MS is a reference table or not.
The function should always be called with a table argument. The ability of omitting the argument is only present for backward compatibility.
The function can be useful to get the correct values from the result of a getcol or getcolslice on the original table.

Arguments

tab	in	Table to which the row numbers refer Allowed:Record Default: F
nbytes	in	Maximum cache size in bytes Allowed:Int

Returns
Vector of Ints

Example

```
  t:=table("W53.MS")
  t.nrows()
# 1920
  data:=t.getcolslice("Data", [1,1], [1,1])
  data::shape
# [1 1 1920]
  selt:=t.query("ANTENNA1==1")
  selt.nrows()
# 300
  subdata:=data[,,selt.rownumbers(t)]
  subdata::shape
# [300]
```

---

table.setmaxcachesize - Function

Package	**utility**
Module	**table**
Tool	**table**

set maximum cache size for column in the table

Synopsis
setmaxcachesize(columnname, nbytes)

Description
It can sometimes be useful to limit the size of the cache used by a column stored with the tiled storage manager. This function requires some more knowledge about the table system and is not meant for the casual user.

Arguments

columnname	in	Name of column Allowed:String
nbytes	in	Maximum cache size in bytes Allowed:Int

Returns
Bool

Example

```
  t:=table("W53.MS")
  t.nrows()
# 1920
  t.setmaxcachesize ("DATA", 4*1024*1024);
# T
```

---

table.isscalarcol - Function

Package	**utility**
Module	**table**
Tool	**table**

is the specified column scalar?

Synopsis
isscalarcol(columnname)

Description
A column may contain either scalars or arrays in each cell. This tool function tests if the specified column has scalar contents.

Arguments

columnname	in	Name of column Allowed:String

Returns
Bool

Example

```
  table("tcal").isscalarcol("StartTime")
# T
  table("tcal").isscalarcol("Gain")
# F
```

---

table.isvarcol - Function

Package	**utility**
Module	**table**
Tool	**table**

tell if column contains variable shaped arrays

Synopsis
isvarcol(columnname)

Description
This functions tells if the column contains variable shaped arrays. If so, the function `getvarcol` should be used to get the entire column. Otherwise `getcol` can be used.

Arguments

columnname	in	Name of column Allowed:String

Returns
Bool

---

table.coldatatype - Function

Package	**utility**
Module	**table**
Tool	**table**

return the column data type

Synopsis
coldatatype(columnname)

Description
A column may contain various data types. This tool function returns the type of the column as a string.

Arguments

columnname	in	Name of column Allowed:String

Returns
String

Example

```
  table("tcal").coldatatype("StartTime")
# double
  table("tcal").coldatatype("Gain")
# complex
```

---

table.colarraytype - Function

Package	**utility**
Module	**table**
Tool	**table**

return the column array type

Synopsis
colarraytype(columnname)

Description
The possible column array types are defined as:

FixedShape FixedShape means that the shape of the array must be the same in each cell of the column. If not given, the array shape may vary. Option Direct forces FixedShape.

Direct Direct means that the data is directly stored in the table. Direct forces option FixedShape. If not given, the array is indirect, which implies that the data will be stored in a separate file.

Arguments

columnname	in	Name of column Allowed:String

Returns
String

Example

```
  table("tcal").colarraytype("Gain")
# Direct,FixedShape
```

---

table.ncols - Function

Package	**utility**
Module	**table**
Tool	**table**

return number of columns

Synopsis
ncols()

Returns
Int

Example

```
  table("tcal").ncols()
# 9
```

---

table.nrows - Function

Package	**utility**
Module	**table**
Tool	**table**

return number of rows

Synopsis
nrows()

Description
Note that rows are numbered starting at 1.

Returns
Int

Example

```
  table("tcal").nrows()
# 11
```

---

table.addrows - Function

Package	**utility**
Module	**table**
Tool	**table**

add a specified number of rows

Synopsis
addrows(nrow)

Description
Rows can be added to the end of a table that was opened readonly=F. The new rows are empty.

Arguments

nrow	in	Number of rows to add Allowed:Int Default: 1

Returns
Bool

---

table.removerows - Function

Package	**utility**
Module	**table**
Tool	**table**

remove the specified rows

Synopsis
removerows(rownrs)

Description
Remove the row numbers specified in the vector from the table. It fails when the table does not support row removal.

Arguments

rownrs	in	Row numbers to remove Allowed:Int

Returns
Bool

---

table.addcols - Function

Package	**utility**
Module	**table**
Tool	**table**

add one or more columns

Synopsis
addcols(desc, dminfo)

Description
Columns can be added to a table that was opened readonly=F. The new columns will be filled with a default value (0 or blank).
For each column to be added a column description has to be setup using function tablecreatescalarcoldesc or tablecreatearraycoldesc. When multiple columns are used, they have to be combined in a single record using tablecreatedesc.
It is possible to specify data manager info in order to define a data manager (storage manager or virtual column engine) for the columns to be added.

Arguments

desc	in	Description of one or more columns Allowed:Descriptor
dminfo	in	Optional description data manager to use Allowed:Record Default: F

Returns
Bool

Example

```
  t:=table("mytable", readonly=F)
  dc3:=tablecreatescalarcoldesc('C3', 'a')
  dc4:=tablecreatescalarcoldesc('C4', as_float(0))
  dc5:=tablecreatearraycoldesc('C5', as_double(0), 2, [10,20])
  t.addcols(dc3)
# T
  t.addcols(tablecreatedesc(dc4, dc5))
# T
```

A single column can be added as such, but multiple columns have to be combined.

---

table.renamecol - Function

Package	**utility**
Module	**table**
Tool	**table**

rename a column

Synopsis
renamecol(oldname, newname)

Description
A column can be renamed in a table that was opened readonly=F.
However, renaming is not possible in a (reference) table resulting from a select or sort operation.

Arguments

oldname	in	name of column to be renamed Allowed:String
newname	in	new name of column Allowed:String

Returns
Bool

Example

```
  t:=table("mytable", readonly=F)
  t.renamecol ('colold', 'colnew')
# T
```

Column `colold` is renamed to `colnew`.

---

table.removecols - Function

Package	**utility**
Module	**table**
Tool	**table**

remove one or more columns

Synopsis
removecols(columnnames)

Description
Columns can be removed from a table that was opened readonly=F.
It may not always be possible to remove a column, because some data managers do not support column removal. However, if all columns of a data manager are removed, it will always succeed. It results in the removal of the entire data manager (and its possible files).
Note that function getdminfo can be used to find which columns are served by which data manager.

Arguments

columnnames	in	names of columns to be removed Allowed:Vector of Strings

Returns
Bool

Example

```
  t:=table("mytable", readonly=F)
  t.removecols ("col1 col2")
# T
```

Two columns are removed.

---

table.iscelldefined - Function

Package	**utility**
Module	**table**
Tool	**table**

test if a specific cell contains a value

Synopsis

iscelldefined(columnname, rownr)

Description
A column containing variable shaped arrays can have an empty cell (if no array has been put into it). This function tests if a cell is defined (thus is not empty). Note that a scalar column and a fixed shape array column cannot have empty cells.

Arguments

columnname	in	Name of column Allowed:String
rownr	in	Row number, starting at 1 Allowed:Int

Returns
Bool

---

table.getcell - Function

Package	**utility**
Module	**table**
Tool	**table**

get a specific cell

Synopsis
getcell(columnname, rownr)

Description
A cell is the value at one row in one column. It may be a scalar or an array.

Arguments

columnname	in	Name of column Allowed:String
rownr	in	Row number, starting at 1 Allowed:Int

Returns
Value

---

table.getcellslice - Function

Package	**utility**
Module	**table**
Tool	**table**

get a slice from a specific cell

Synopsis
getcellslice(columnname, rownr, blc, trc, incr)

Description
A cell is the value at one row in one column. It must be an array. The slice must be specified as blc, trc with an optional stride.
In blc and trc -1 can be used to indicate all values for a dimension (-1 in blc is equivalent to 1, so -1 is especially useful for trc).

Arguments

columnname	in	Name of column Allowed:String
rownr	in	Row number, starting at 1 Allowed:Int
blc	in	Bottom left corner (e.g. [1,1,1] is start of 3D array) Allowed:Vector of Ints
trc	in	Top right corner Allowed:Vector of Ints
incr	in	Stride (defaults to 1 for all axes) Allowed:Vector of Ints

Returns
Array

Example

```
  t:=table("W53.MS")
  data:=t.getcellslice("DATA", 1, [1,1], [2,1])
  data::shape
# [2 1]
```

---

table.getcol - Function

Package	**utility**
Module	**table**
Tool	**table**

get a specific column

Synopsis
getcol(columnname, startrow, nrow, rowincr)

Description
The entire column (or part of it) is returned. Warning: it might be big! The functions can only be used if all arrays in the column have the same shape. That is guaranteed for columns containing scalars or fixed shaped arrays. For columns containing variable shaped arrays it only succeeds if all those arrays happen to have the same shape.
Note that function `getvarcol` can be used to get a column of arbitrary shaped arrays, which also handles empty cells correctly. Function `isvarcol` tells if a column contains variable shaped arrays. shaped

Arguments

columnname	in	Name of column Allowed:String
startrow	in	First row to read (default 1) Allowed:Int
nrow	in	Number of rows to read (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to read (default 1) Allowed:Int

Returns
Array

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  gain:=t.getcol("Gain")
  gain::shape
# [2 2 28 1 11]
```

---

table.getvarcol - Function

Package	**utility**
Module	**table**
Tool	**table**

get a specific column (for variable arrays)

Synopsis
getvarcol(columnname, startrow, nrow, rowincr)

Description
Function `getcol` can only used if values in the column cells to get have the same shape. Function `getvarcol` addresses this limitation by returning the values as a record instead of an array. Each field in the record contains the value for a column cell. If the value is undefined (i.e. the cell does not contain a value), the unset value is put in the record. Each field name is the letter r followed by the row number. The length of the record is the number of rows to get.
Note that the function `isvarcol` tells if a column contains variable shaped arrays.

Arguments

columnname	in	Name of column Allowed:String
startrow	in	First row to read (default 1) Allowed:Int
nrow	in	Number of rows to read (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to read (default 1) Allowed:Int

Returns
Record

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  gain:=t.getvarcol("Gain")
  length(gain)
# 11
```

---

table.getcolslice - Function

Package	**utility**
Module	**table**
Tool	**table**

get a slice from a specific column

Synopsis
getcolslice(columnname, blc, trc, incr, startrow, nrow, rowincr)

Description
A slice from the entire column (or part of it) is returned. Warning: it might be big!
In blc and trc -1 can be used to indicate all values for a dimension (-1 in blc is equivalent to 1, so -1 is especially useful for trc). Note that blc and trc should not contain the row number, only the blc and trc of the arrays in the column.

Arguments

columnname	in	Name of column Allowed:String
blc	in	Bottom left corner (e.g. [1,1,1] is start of 3D array) Allowed:Vector of Ints
trc	in	Top right corner Allowed:Vector of Ints
incr	in	Stride (defaults to 1 for all axes) Allowed:Vector of Ints
startrow	in	First row to read (default 1) Allowed:Int
nrow	in	Number of rows to read (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to read (default 1) Allowed:Int

Returns
Array

Example

```
  t:=table("W53.MS")
  data:=t.getcolslice("DATA", [1,1], [2,1])
  data::shape
# [2 1 1920]
```

---

table.putcell - Function

Package	**utility**
Module	**table**
Tool	**table**

put a specific cell

Synopsis
putcell(columnname, rownr, value)

Description
A cell is the the value at one row in one column. It may be a scalar or an array.

Arguments

columnname	in	Name of column Allowed:String
rownr	in	Row number(s) (1-relative) Allowed:Vector of Ints
value	in	Value

Returns
Bool

---

table.putcellslice - Function

Package	**utility**
Module	**table**
Tool	**table**

put a slice into a specific cell

Synopsis
putcellslice(columnname, rownr, value, blc, trc, incr)

Description
A cell is the value at one row in one column. It must be an array. The slice must be specified as blc, trc with an optional stride.
In blc and trc -1 can be used to indicate all values for a dimension (-1 in blc is equivalent to 1, so -1 is especially useful for trc).

Arguments

columnname	in	Name of column Allowed:String
rownr	in	Row number, starting at 1 Allowed:Int
value	in	Value
blc	in	Bottom left corner (e.g. [1,1,1] is start of 3D array) Allowed:Vector of Ints
trc	in	Top right corner Allowed:Vector of Ints
incr	in	Stride (defaults to 1 for all axes) Allowed:Vector of Ints

Returns
Bool

---

table.putcol - Function

Package	**utility**
Module	**table**
Tool	**table**

put a specific column

Synopsis
putcol(columnname, value, startrow, nrow, rowincr)

Arguments

columnname	in	Name of column Allowed:String
value	in	Array
startrow	in	First row to put (default 1) Allowed:Int
nrow	in	Number of rows to put (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to put (default 1) Allowed:Int

Returns
Bool

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  gain:=t.getcol("Gain")
  gain[1,1,1,6,]:=1+0i
  t.putcol("Gain", gain)
  t.flush()
```

---

table.putvarcol - Function

Package	**utility**
Module	**table**
Tool	**table**

put a specific column (for variable arrays)

Synopsis
putvarcol(columnname, value, startrow, nrow, rowincr)

Description

`putcol` can only used if values in the column cells to put have the same shape. `putvarcol` addresses this limitation by passing the values as a record instead of an array. Each field in the record contains the value for a column cell. So the length of the record has to match the number of rows to put. If a value is the unset value, no put is done for that row.

Arguments

columnname	in	Name of column Allowed:String
value	in	Record with values
startrow	in	First row to put (default 1) Allowed:Int
nrow	in	Number of rows to put (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to put (default 1) Allowed:Int

Returns
Bool

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  gain:=t.getvarcol("DATA", 1, 10)
  t.putvarcol("Gain", gain, 11, 10)
  t.flush()
```

This example copies the values from row 1-10 to row 11-20.

---

table.putcolslice - Function

Package	**utility**
Module	**table**
Tool	**table**

put a slice into a specific column

Synopsis
putcolslice(columnname, value, blc, trc, incr, startrow, nrow, rowincr)

Description

In blc and trc -1 can be used to indicate all values for a dimension (-1 in blc is equivalent to 1, so -1 is especially useful for trc). Note that blc and trc should not contain the row number, only the blc and trc of the arrays in the column.

Arguments

columnname	in	Name of column Allowed:String
value	in	Array
blc	in	Bottom left corner (e.g. [1,1,1] is start of 3D array) Allowed:Vector of Ints
trc	in	Top right corner Allowed:Vector of Ints
incr	in	Stride (defaults to 1 for all axes) Allowed:Vector of Ints
startrow	in	First row to put (default 1) Allowed:Int
nrow	in	Number of rows to put (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to put (default 1) Allowed:Int

Returns
Bool

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  gain:=t.getcolslice("Gain", [1,1,1,6,1], [1,1,1,6,20])
  gain:=1+0i
  t.putcolslice("Gain", gain, [1,1,1,6,1], [1,1,1,6,20])
  t.flush()
```

---

table.getcolshapestring - Function

Package	**utility**
Module	**table**
Tool	**table**

get shape of arrays in a specific column

Synopsis

getcolshapestring(columnname, startrow, nrow, rowincr)

Description

The shapes of the arrays in the entire column (or part of it) are returned as strings like [20,3]. When the column contains fixed shaped arrays, a single string is returned. Otherwise a vector of strings is returned.

Arguments

columnname	in	Name of column Allowed:String
startrow	in	First row to read (default 1) Allowed:Int
nrow	in	Number of rows to read (default -1 means till the end) Allowed:Int
rowincr	in	Increment in rows to read (default 1) Allowed:Int

Returns

Array of Strings

Example

```
  t:=table("tcal",readonly=F)
# successful read/write open of table  tcal :  9 columns, 11 rows
  shapes:=t.getcolshapestring("Gain")
```

---

table.getkeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

get value of specific table keyword

Synopsis
getkeyword(keyword)

Description
The value of the given table keyword is returned. The value can be of any type, including a record and a table.
If a keyword is a table, its value is returned as a string containing the table name prefixed by 'Table: '. That prefix is automatically removed by table functions like `open` and `tableexists`.
It is possible that the value of a keyword is a record itself (arbitrarily deeply nested). A field in such a subrecord can be read by separating the name with dots.

Arguments

keyword	in	Name or seqnr of keyword Allowed:String—Int

Returns
Value

Example

```
  t:=table('tcal')
# successful readonly open of table  tcal :  9 columns, 33 rows
  t.getkeyword('Interval')
# 30
  t.getkeyword('rec.fld')     # get field from a record
# 3.14
```

---

table.getkeywords - Function

Package	**utility**
Module	**table**
Tool	**table**

get values of all table keywords

Synopsis
getkeywords()

Description
The values of all table keywords are returned. The values can be of any type, including a record and a table.
If a keyword is a table, its value is returned as a string containing the table name prefixed by 'Table: '. That prefix is automatically removed by table functions like `open` and `tableexists`.

Returns
Record

Example

```
  t:=table('tcal')
# successful readonly open of table  tcal :  9 columns, 33 rows
  t.getkeywords()
# [Type=T Jones, Interval=30, DeltaT=1]
```

---

table.getcolkeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

get value of specific column keyword

Synopsis
getcolkeyword(columnname, keyword)

Description
The value of the given column keyword is returned. The value can be of any type, including a record and a table.
If a keyword is a table, its value is returned as a string containing the table name prefixed by 'Table: '. That prefix is automatically removed by table functions like `open` and `tableexists`.
It is possible that the value of a keyword is a record itself (arbitrarily deeply nested). A field in such a subrecord can be read by separating the name with dots.

Arguments

columnname	in	Name of column Allowed:String
keyword	in	Name or seqnr of keyword Allowed:String—Int

Returns
Value

Example

```
  t:=table("MS")
# successful readonly open of table  MS :  22 columns, 7669 rows
  t.getcolkeyword("UVW", "UNIT")
# m
```

---

table.getcolkeywords - Function

Package	**utility**
Module	**table**
Tool	**table**

get values of all keywords for a column

Synopsis
getcolkeywords(columnname)

Description
The values of all keywords for the given column are returned. The values can be of any type, including a record and a table.
If a keyword is a table, its value is returned as a string containing the table name prefixed by 'Table: '. That prefix is automatically removed by table functions like `open` and `tableexists`.

Arguments

columnname	in	Name of column Allowed:String

Returns
Record

Example

```
  t:=table("MS")
# successful readonly open of table  MS :  22 columns, 7669 rows
  t.getcolkeywords("UVW")
# [UNIT=m, MEASURE_TYPE=UVW, MEASURE_REFERENCE=, _ForwardColumn_TableName=MS]
```

---

table.putkeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

put a specific table keyword

Synopsis
putkeyword(keyword, value, makesubrecord)

Description

Put a table keyword. The value of the keyword can be a scalar or an array of any type or it can be a record.
It is possible to define a keyword holding a subtable. In that case a special string containing the name of the subtable will be passed to the table client.
It is possible that the value of a keyword is a record itself (arbitrarily deeply nested). A field in such a subrecord can be written by separating the name with dots. If a subrecord does not exist, an error is returned unless `makesubrecord=T` is given. In such a case intermediate records are created when needed.

Arguments

keyword	in	Name or seqnr of keyword Allowed:String—Int
value	in	Value of keyword
makesubrecord	in	Create intermediate records Allowed:Bool Default: F

Returns

Bool

Example

```
  t:=table("MS", readonly=F)
  t.putkeyword("VERSION", "1.66")
# T
#      define ANTENNA subtable
  t.putkeyword("ANTENNA", 'Table: MS/ANTENNA')
  t.flush()
# T
#      write a field in a record and create subrecords when needed
  t.putkeyword("REC.SUB.FLD", "val", T)
# T
#      write a keyword with a record value
  t.putkeyword("REC", [SUB=[FLD="val"]])
# T
```

Note that the last example does the same as the previous one (assuming that `REC` does not exist yet with other fields).

---

table.putkeywords - Function

Package	**utility**
Module	**table**
Tool	**table**

put multiple table keywords

Synopsis
putkeywords(value)

Description
Put multiple table keywords. All fields in the given record are put as table keywords. The value of each field can be a scalar or an array of any type or it can be a record.
It is also possible to define a keyword holding a subtable. This can be done by giving the keyword a string value consisting of the subtable name prefixed by 'Table: '. Remember to use single quotes, otherwise glish turns it into a string vector.

Arguments

value	in	Record of keyword=value pairs

Returns
Bool

Example

```
  t:=table('tcal', readonly=F)
  kw:=t.getkeywords()
  print kw
# [Type=T Jones, Interval=30, DeltaT=1]
  kw.Interval:=45
  t.putkeywords(kw)
# T
  t.flush()
# T
```

---

table.putcolkeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

put a specific keyword for a column

Synopsis
putcolkeyword(columnname, keyword, value)

Description
Put a keyword in the given column. The value of the keyword can be a scalar or an array of any type or it can be a record.
It is possible to define a keyword holding a subtable. In that case a special string containing the name of the subtable will be passed to the table client.
It is possible that the value of a keyword is a record itself (arbitrarily deeply nested). A field in such a subrecord can be written by separating the name with dots. If a subrecord does not exist, an error is returned unless `makesubrecord=T` is given. In such a case intermediate records are created when needed.

Arguments

columnname	in	Name of column Allowed:String
keyword	in	Name or seqnr of keyword Allowed:String—Int
value	in	Value of keyword

Returns
Bool

Example

```
  t:=table("MS", readonly=F)
  t.putcolkeyword("UVW", "MEASURE_UNIT", "m")
# T
  t.flush()
# T
```

---

table.putcolkeywords - Function

Package	**utility**
Module	**table**
Tool	**table**

put multiple keywords for a column

Synopsis
putcolkeywords(columnname, value)

Description
Put multiple keywords in the given column. All fields in the given record are put as column keywords. The value of each field can be a scalar or an array of any type or it can be a record.
It is also possible to define a keyword holding a subtable. This can be done by giving the keyword a string value consisting of the subtable name prefixed by 'Table: '. Remember to use single quotes, otherwise glish turns it into a string vector.

Arguments

columnname	in	Name of column Allowed:String
value	in	Record of keyword=value pairs

Returns
Bool

Example

```
  t:=table("MS", readonly=F)
  t.getcolkeywords("UVW")
  kw:=[UNIT=m, MEASURE_TYPE=UVW, MEASURE_REFERENCE=, _ForwardColumn_TableName=MS]
  kw.MEASURE_REFERENCE='B1950'
  t.putcolkeywords(kw)
# T
```

table.removekeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

remove a specific table keyword

Synopsis
removekeyword(keyword)

Arguments

keyword	in	Name or seqnr of keyword Allowed:String—Int

Returns
Bool

Example

```
  t:=table("MS", readonly=F)
  t.removekeyword("VERSION")
# T
  t.flush()
# T
```

---

table.removecolkeyword - Function

Package	**utility**
Module	**table**
Tool	**table**

remove a specific keyword for a column

Synopsis
removecolkeyword(columnname, keyword)

Arguments

columnname	in	Name of column Allowed:String
keyword	in	Name or seqnr of keyword Allowed:String—Int

Returns
Bool

Example

```
  t:=table("MS", readonly=F)
  t.removecolkeyword("UVW", "MEASURE_UNIT")
# T
  t.flush()
# T
```

---

table.getdminfo - Function

Package	**utility**
Module	**table**
Tool	**table**

get the info about data managers

Synopsis
getdminfo()

Description
This function returns the types and names of the data managers used. For each data manager it also returns the names of the columns served by it. The information is returned as a record containing a subrecord for each data manager. Each subrecord contains the fields TYPE, NAME and COLUMNS.

Returns
record

Example

```
  t1:=table('/aips++/code/aips/implement/Tables/test/tTable_2.data_v0')
# successful readonly open of default-locked table
#   /aips++/code/aips/implement/Tables/test/tTable_2.data_v0:
#   9 columns, 10 rows
  t.getdminfo()
# [*1=[TYPE=StandardStMan, NAME=SSM, COLUMNS=ac ad ag arr1 arr2 arr3],
#  *2=[TYPE=StManAipsIO, NAME=, COLUMNS=ab],
#  *3=[TYPE=IncrementalStMan, NAME=ISM, COLUMNS=ae af]]
```

This shows that the table uses 3 storage managers.

---

table.keywordnames - Function

Package	**utility**
Module	**table**
Tool	**table**

get the names of all table keywords

Synopsis
keywordnames()

Description
This function returns a vector of strings containing the names of all table keywords.

Returns
Vector of Strings

---

table.fieldnames - Function

Package	**utility**
Module	**table**
Tool	**table**

get the names of fields in a table keyword

Synopsis
fieldnames(keyword)

Description
This function returns a vector of strings containing the names of all fields in the given table keyword. It is only valid if the keyword value is a record.
If no keyword name is given, the names of all table keywords are returned.

Arguments

keyword	keyword name Allowed:String Default: ”

Returns
Vector of Strings

---

table.colkeywordnames - Function

Package	**utility**
Module	**table**
Tool	**table**

get the names of all keywords in a column

Synopsis
colkeywordnames(columnname)

Description
This function returns a vector of strings containing the names of all keywords in the column with the given name..

Arguments

columnname	column name Allowed:String

Returns
Vector of Strings

table.colfieldnames - Function

Package	**utility**
Module	**table**
Tool	**table**

get the names of fields in a keyword in a column

Synopsis
colfieldnames(columnname, keyword)

Description
This function returns a vector of strings containing the names of all fields in the given keyword in the given column. It is only valid if the keyword value is a record.
If no keyword name is given, the names of all keywords in the column are returned.

Arguments

columnname	column name Allowed:String
keyword	keyword name Allowed:String Default: ”

Returns
Vector of Strings

table.getdesc - Function

Package	**utility**
Module	**table**
Tool	**table**

get the table description

Synopsis

getdesc(actual)

Description

The table description is a Glish record that contains a complete description of the layout of the table (except for the number of rows). It can be used as a template in the creation of another table using the table constructor.
By default the actual table description is returned (thus telling the actual shapes and data managers used). It is also possible to get the table description used when creating the table.

Arguments

actual	in	actual table description? Allowed:Bool Default: T

Returns

Table description record

Example

```
  t:=table('tcal')
# successful readonly open of table  tcal :  9 columns, 33 rows
  t.getdesc()
# [StartTime=[valueType=double, dataManagerType=StandardStMan,
#     dataManagerGroup=StandardStman, option=5, maxlen=0, comment=],
#  StopTime=[valueType=double, dataManagerType=StandardStman,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=],
#  Gain=[valueType=complex, dataManagerType=StandardStMan,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=,
#      ndim=4, shape=[2 2 28 1] ],
#  SolutionOK=[valueType=boolean, dataManagerType=StandardStman,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=],
#  Fit=[valueType=float, dataManagerType=StandardStMan,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=],
#  FitWeight=[valueType=float, dataManagerType=StandardStMan,
#      dataManagerGroup=StandardStman, option=5, maxlen=0, comment=],
#  iSolutionOK=[valueType=boolean, dataManagerType=StandardStman,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=,
#      ndim=2, shape=[28 1] ],
#  iFit=[valueType=float, dataManagerType=StandardStMan,
#      dataManagerGroup=StandardStman, option=5, maxlen=0, comment=,
```

```
#      ndim=2, shape=[28 1] ],
#  iFitWeight=[valueType=float, dataManagerType=StandardStMan,
#      dataManagerGroup=StandardStMan, option=5, maxlen=0, comment=,
#      ndim=2, shape=[28 1] ]]
```

---

table.getcoldesc - Function

Package	**utility**
Module	**table**
Tool	**table**

get the description of a specific column

Synopsis
getcoldesc(columnname)

Description
The column description is a Glish record that contains a complete description of the layout of a specified column (except for the number of rows). It can be used to construct a table description.

Arguments

columnname	in	Name of column Allowed:String

Returns
Column description record

Example

```
  t:=table('tcal')
# successful readonly open of table  tcal :  9 columns, 33 rows
  t.getcoldesc('Gain')
# [valueType=complex, dataManagerType=StandardStMan, dataManagerGroup=StandardStMan, option=5, comment=, ndim=4, shape=[2 2 28 1] ]
```
