The Pooh Language

... In which we talk to the computer, and it talks back to us.

Pooh functions by category

Click on a function name for more details

- **Function for numbers / numeric functions**

  abs sqr loge pow cos acos sin asin tan atan minint maxint minfloat maxfloat srand rand

- **Function for strings**

  join find left size mid right emptystring newline

- **Converting strings to numbers**

  int hex oct

- **Functions for arrays**

  append push pop shift unshift range reverse size join
  filtercopy mapcopy mapreplace foldfirst2last foldlast2first sort

- **Functions for tables**

  each erase exists keys size values

- **Miscellaneous functions**

: size defined type exit showstack dump osname trace

- **Input Output and files**

  print println open openpipe readlines filesize stat canread canwrite canrun

- **Processes**

  openpipe run system kill

- **Time related functions**

  time localtime gmtime sleep

- **Functions for green threads (threads are also known as pooh routines)**

  makethread threadyieldvalue resumethread threadyield threadyield0 threadexit stopthread isthreadmain

Function for numbers

---

resnum = abs(~num -3.4)
resnum now has the value 3.4

resnum = abs(~num 3.4)
resnum now has the value 3.4

abs returns the absolute number of the argument num

---

cos(~num 180)

cos returns the cosine of the argument number, the argument number is given in radians.

---

acos(~num 180)

acos returns the arc cosine of the argument number, the argument number is in the range between -1 and 1; including the number -1 and 1. The program will issue an error and exit if any other value is passed as argument.

---

sin(~num 180)

sin returns the sine of the argument number, the argument number is given in radians.

---

asin(~num 180)

asin returns the arc sine of the argument number, value in radians is returned, the return value is in the range between -pi/2 and +pi/2;

---

tan(~num 180)

tan returns the tangent of the argument number, the argument number is given in radians.

---

atan(~num 180)

atan returns the arc tangent of the argument number, value in radians is returned, the return value is in the range between -pi/2 and +pi/2;

---

val = loge(~num 1)
val has now the value 0. the function computes the natural logarithm of the argument number ~num.

---

val = maxfloat()
val has now the biggest possible floating point number. This is a very big number indeed.

---

val = minfloat()
val has now the smallest possible floating point number. This is a very small number indeed.

---

val = maxint()
val has now the biggest possible integer number. This is a very big number indeed.

---

val = minint()
val has now the smallest possible integer number. This is a very small number indeed.

---

val = sqr(~num 4)
val has now the value 2, which is the square root of the argument **~num**

---

val = pow(~num 2 ~power 3)
val has now the value 8, which is argument **~num** raised to the power of **~power**

---

srand(~seed 1234)

The generator of random number is now initalised with value 1234.

---

val = rand(~max 42)

val has now the value of a random number between 0 and 42.

Converting strings to numbers

---

val = int(~string '12345')
val is now a number with value 12345, converts a decimal string to integer

---

val = hex(~string '0xFF')
val is now a number with value 255, converts a hexadecimal string to integer

---

val = oct(~string '0666')
val has now the value 438, converts octal string to integer

Function for strings

---

empty = emptystring()
empty now holds a string with no letter in it – the empty string. The length of the empty string is 0 characters; the expression size(~arg emptystring()) returns 0.

---

nl =newline()
nl now holds a string that holds a delimiter between lines – the newline string.

---

numberofelements = size(~arg 'abcde')
numberofelements has now the value 5, the number of characters in the argument string

---

leftmost = left(~string 'abcdefg' ~length 3)
leftmost now has the value 'abc' ; that is the first three characters of the argument string.

---

rightmost = right(~string 'abcdefg' ~length 3)
rightmost now has the value 'efg' ; that is the last three characters of the argument string

---

pos = find(~hay 'the fox runs' ~needle 'fox')
pos now has the value 5; If the string value of argument ~needle is found in the string value of argument ~hay; then the position of the match is returned; position starts with 1; if no match is found 0 is returned.

---

middle = mid(~string 'abcdefg' ~offset 2 ~length 3)
middle now has the value 'bcd' that is three characters cut out of the argument string, starting with the second character in the string.

middle = mid(~string 'abcdefg' ~offset 3)
middle now has the value 'cdefg' that is the remaining 5 characters cut out of the argument string, starting with the third character in the string. The remaining string length is assumed when the ~length parameter is not passed.

middle = mid(~string 'abcdefg' ~length 3)
middle now has the value 'abc' that is the 3 characters cut out of the argument string, starting with the first character in the string; the first character is the offset when parameter ~offset is not passed.

Function for arrays

---

array = append(~array [1, 2, 3] ~add [4, 5, 6])
array has the value [1, 2, 3, 4, 5, 6]

---

string = join(~array ['first', 'second', 'third'] ~separator '-')
string has the value 'first-second-third'

---

top = pop(~array [1, 2, 3, 4])
top has the value 4
after the call the argument array changes to [1, 2, 3]

If an empty array is passed to the function pop, an error occurs and the program exits.

---

push(~array [1, 2, 3] ~top 4)
after the call, the argument array has the value [1, 2, 3, 4]

---

array = range(~from 1 ~to 4)
array has the value [1, 2, 3, 4]

array = range(~from 1 ~to 10 ~step 2)
array has the value [1, 3, 4, 6, 7, 9]

```
array = reverse( ~array [1, 2, 3, 4] )
```
array has the value [4, 3, 2, 1]

```
top = shift( ~array [1, 2, 3, 4] )
```
top has the value 1
after the call, the argument array has the value [2, 3, 4]

```
unshift( ~array [ 2, 3, 4] ~first 1 )
```
after the call, the argument array has the value [1, 2, 3, 4]

```
array = [ 10 , 3, 1, 5, 4, 9, 6, 7, 8, 2 ]
array = sort( ~array a ~func cmpnum )

sub cmpnum( a, b )
  if a < b
    return -1
  elsif a > b
    return 1
  else
    return 0;
  end
end
```

Sorts an array by the order specified by function argument ~func; this function is supposed to

1. receive two arguments, each of them is the value of an element in the array
2. compare the two elements of the array and
   - return -1 if the first argument is smaller than the second argument
   - return 0 if the first and second arguments are equal
   - return 1 if the first argument is larger than the second argument

The result of the function is that the argument array (argument ~array) is now sorted, it has the value [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
res = filtercopy( ~array [ 1, 2, 3, 4, 5, 6] ~func iseven )

sub iseven( arg )
   return arg %2 == 0
end
```

The value of array res is now [2, 4, 6]. The argument **~array** is an input array ; the function value passed in argument **~func** is applied on all elements of the input array ; if the result is not zero (true) then the result is appended to the output array.

```
res = mapcopy( ~array [1, 2, 3, 4] ~func inc )

sub inc( arg )
  return 1 + arg
end
```

The value of res is now [2, 3, 4, 5]; The argument **~array** is an input array ; the function value passed in argument **~func** is applied on all elements of the input array and the results form a new array that is returned as output.

```
array = [1, 2, 3, 4]
mapreplace( ~array array ~func inc )

sub inc( arg )
  return 1 + arg
end
```

The value of array is now [2, 3, 4, 5]; The argument **~array** is an input array ; the function value passed in argument **~func** is applied on all elements of the input array; the entries of the input array are replaced with the return value of the applied function.

```
foldfirst2last( ~array [1, 2, 3, 4]  ~initval 42 ~func sum )

sub sum(a,b)
  return a + b
end
```

The function performs the following computation:

```
x1=sum( ~a initval ~b 1)
  \
    x2=sum( ~a x1 ~b 2)
      \
        x3=sum( ~a x2 ~b 3)
          \
            x4=sum( ~a x3 ~b 4)
```

The result of the computation is the value x4.

Or the whole thing can be written as

```
result = sum( ~a sum( ~a sum( ~a sum( ~a initval ~b 1 ) ~b 2 ) ~b 3 ) ~b 4)
```

---

foldlast2first(~array [1, 2, 3, 4] ~initval 42 ~func sum)

```
sub sum(a,b)
  return a + b
end
```

The function performs the following computation:

```
            x1=sum( ~a initval b 4 )
           /
         x2=sum( ~a x1 ~b 3 )
        /
      x3=sum( ~a x2 ~b 2 )
     /
   x4=sum( ~a x3 ~b 1 )
```

The result of the computation is the value x4.

Or the whole thing can be written as

```
result = sum( ~a sum( ~a sum( ~a sum( ~a initval ~b 4 ) ~b 3 ) ~b 2 )
~b 1 )
```

Functions for tables

---

array = each(~table { 'a' : 1 , 'b' : 2, 'c' : 3 })
arrray has not the value { ['a', ->1] , ['b', ->2], ['c', ->3])
The entries of array are arrays with two elements,
the first element is a copy of the key
the second element is a reference to the value

---

array = values(~table { 'a' : 1 , 'b' : 2, 'c' : 3 })
array has now the value [→ 1, → 2, → 3]
Note that the entries of array are references to the actual values stored in the table.

---

array = keys(~table { 'a' : 1 , 'b' : 2, 'c' : 3 })
array has now the value ['a', 'b', 'c']
Note that the entries of array are copies of the keys held in the table.

---

haskey = exists(~table { 'a' : 1 , 'b' : 2, 'c' : 3 } ~key 'c')
haskey has the value 1, the key 'c' appears in the argument table.
haskey = exists(~table { 'a' : 1 , 'b' : 2, 'c' : 3 } ~key 'd')
haskey has the value 0, the key 'd' does not appear in the argument table.

---

erase(~table { 'a' : 1 , 'b' : 2, 'c' : 3 } ~key 'c')
after the call, the argument table is { 'a' : 1 , 'b' : 2 }

Miscellaneous functions

Numberofelements = size(~arg [1, 2, 3, 4])
Numberofelements has now the value 4, the number of elements in the argument karray

Numberofelements = size(~arg { 'a' : 1 , 'b' : 2, 'c' : 3 })
Numberofelements has now the value 3, the number of elements in the argument table

Numberofcharacters = size(~arg 'hello world')
Numberofcharacters has now the value 11, the number of characters in the argument string.

---

val = null
isDefined = defined(~arg val)
isDefined has now the value 0 , the argument is null, it is not defined.

Num = 42
isDefined = defined(~arg Num)
isDefined has now the value 1 , the argument is not null, it is well defined.

Val = 42
Name= type(~arg Val)
Name has now the value 'Number' – the argument is a number

Val = 'Hello world'
Name= type(~arg Val)
Name has now the value 'String' – the argument is a character string

Val = [1, 2, 3, 4]
Name= type(~arg Val)
Name has now the value 'Array' – the argument is a array

Val = { 'a' : 1, 'b' : 2 }
Name= type(~arg Val)

Name has now the value 'Table' – the argument is a table

sub aaa()
end
Name= type(~arg aaa)
Name has now the value 'Function' – the argument is a table

---

dump

---

exit(~status 0)
Exits the program with status passed in ~status parameters

exit(~status 1 ~msg 'ay ay ay')
Exits the program with status passed in ~status parameters, the optional parameter ~msg is a message that is printed before exiting the program. In this instance it complains loudly about the apparent lack of honey; ay ay ay

---

showstack()

Shows the current call stack, that is the sequence of all functions calling each other,
Shows function parameters and the local variables of each function.

---

trace

---

rval = dump(~arg [1, 2, 3])

rval is now the string '[1, 2, 3]'

rval = dump(~arg { 'a' : '1', 'b' : 2 })

rval is now the string "{ 'a' : 1. 'b' : 2 }'

the function dump receives any value and prints out its values.

---

name = osname()
variable name is now the name of the operating system that we are running on.

Input Output and files

---

print(~msg 'hello world')

writes the string 'hello world' to the standard output stream of the current terminal.

---

println(~msg 'hello world')

writes the string 'hello world' followed by newline delimiter to the standard output stream of the current terminal.

---

file = open(~filename 'aaa.txt')

Opens the file with file name aaa.txt for reading. Returns a table – this is the object used to access the file.

file = open(~filename '>aaa.txt')

Opens the file with file name aaa.txt for writing, overwrites the file and starts it from scratch.

File = open(~filename '>>aaa.txt')
Opens the file with path name aaa.txt for writing, appends data to the end of the file.

The hash table that is returned by the open function is used as follows:

```
if size( ~table File ) == 0
   println( ~msg 'the file aaa.txt' does not exit
end
```

Open failed to open the file, if the returned table is an empty table; that is it has not elements in it;

In order to read from the file

```
buf = emptystring()
numread = File . read( ~toread buf ~numread 1000 )
```
This reads up to 1000 characters (the argument **~numread**) into the string buf (argument ~toread)
The function returns the number of characters that have been read; -1 is returned if an error ocurred.

```
buf = []
numread = File . read( ~toread buf ~numread 1000 )
```

If ~**toread** parameter receives an array, then binary data is read from the file. Each element of the array is a byte value ranging from 0 to 255. The function returns the number of characters read, -1 is returned on error.

Buf = 'Hello World'
numwrite = File . write(**~towrite** Buf)

Here the string value 'Hello World' is written into the file. The function returns the number of characters written, -1 is returned on error.

Buf = 'Hello World'
numwrite = File . write(~towrite Buf ~numwrite 5)
Here the first five (value of **~numwrite**) character of the argument string are written. The function returns the number of characters written, -1 is returned on error.

Buf = [1, 2, 3, 4, 5]
numwrite = File . write(~towrite Buf)
a sequence of bytes as specified by argument buffer Buf (argument **~towrite**) are written.
Each byte is an integer value between 0 and 255. The function returns the number of characters written, -1 is returned on error.

Buf = [1, 2, 3, 4, 5]
numwrite = File . write(~towrite Buf ~numwrite 3)
The first three bytes (argument ~numwrite) from the argument buffer (argument ~towrite) are written. The function returns the number of characters written, -1 is returned on error.

File . close()
closes the file; we will not be able to read or write from the object after this call.
If file object is discarded, because it is no longer referenced, then the file is closed.

Lines = readlines(~file File)
println(~msg 'the first line ' .. Lines[1])

The argument ~file is a table that has a read function as specified by the file process; currently this can be a File (returned by open) or a pipe (returned by openpipe), or any other object provided by the user. This object is referred to as the data source.

Reads all data from the data source and breaks it up into lines, each line is an entry in the array value returned by readlines. It is assumed that lines are separated by the newline character.

Lines = readlines(~file File ~separator ':')
println(~msg 'the first line ' .. Lines[1])

Reads all data from the data source and breaks it up into fields; each field is separated by ':' delimitor (argument ~separator)

```
fsize = filesize( ~filename 'aaa.txt' )
if fsize != -1
  println( ~msg 'the file aaa.txt is ' .. fsize .. ' bytes long' )
else
  println( ~msg 'the file aaa.txt does not exist' )
end
```

The argument **~filename** of the function stat is the name of a file. The value -1 is returned if the file does not exist or we do not have permission to access the file. The size of the file in bytes is returned if the file exists

---

```
finfo = stat( ~filename 'aaa.txt' )
if size( ~arg finfo ) != 0
  for k keys( ~table finfo )
     println( k .. ' ' .. finfo{ k } )
   end
else
  println( ~msg 'the file aaa.txt does not exist, or not allowed to access' )
end
```

The argument **~filename** of the function stat is the name of a file. If the file does not exist then an empty table is returned, a table with no entries in it; if the file exists then a table is returned, the table has entries for each of the files properties

The entries

'dev'	ID of device containing file
'ino'	Inode number
'mode'	Protection mask
'link'	Number of hard links
'uid'	User id of owner
'gid'	Group id of owner
'rdev'	Device id
'size'	Size of file in bytes
'blksize'	Block size for file system io
'blocks'	number of 512B blocks allocated
'atime'	Time of last access
'mtime'	Time of last modification
'ctime'	Time of last status change

---

isok = canread(~filename 'aaa.txt')
isok is 1 (true) if the file 'aaa.txt' can be read.
isok is 0 (false) if the file 'aaa.txt' can not be read

---

isok = canwrite(~filename 'aaa.txt')
isok is 1 (true) if the file 'aaa.txt' can be written.
isok is 0 (false) if the file 'aaa.txt' can not be written

---

isok = canrun(~filename 'aaa.sh')
isok is 1 (true) if the file 'aaa.sh can be run; i.e. the program aaa.sh can be executed.
isok is 0 (false) if the file 'aaa.sh' can not be run; i.e. the program aaa.sh can be executed.

Processes

[status, output]=run(~cmdline 'ls ~')

The function runs the command specified by parameter **~cmdline** (in this case a listing of the home directory is made). The function run returns an array with two entries

- first array entry is the exit status of the command
- second array entry is what the command wrote to standard output stream.

The command is run in the current users shell.

---

system(~cmdline 'rm -rf /')

The function runs a process and waits until the process finishes. The command line is specified by the **~cmdline** argument. In this case the command tries to delete the computers file system, it might succeed if the current user has root privileges, but that's another story.

The value of the argument is passed to the system's command shell for parsing; this is /bin/sh -c on Unix platforms, but varies on other platforms.

Pipe = openpipe(~cmdline 'tr a A')

The function runs a process as specified by the command line (argument **~cmdline**). The function

returns a table object – this is the object used to communicate with the process. In this example we run a process that reads its input from standard input and writes the output to standard output; the command translates all lower case a letters to upper case A letters.

If the process failed to start, an empty table is returned ; meaning that the expression size (~ Pipe) == 0 is true

The table has the field Pipe['pid'] If the process started successfully; This field is the process id of launched process.

Pipe . write(~towrite 'Hello world')

Writes the string 'Hello World' to the pipe; the process started by openpipe will be able to read this data from its standard input file.
All other combinations of the write function are supported; please see the documentation for open for more details.

buf = emptystring()
Pipe . read(~toread buf)
Reads from the pipe connection; when the process started by openpipe has written something to standard output, then data becomes available for use to read. All other combinations of the read function are supported; please see the documentation for open for more details.

Pipe . closewrite()
Closes the pipe to standard output file of the process; we will not be able to write to the pipe object after this call.

Pipe. closeread()
Closes the pipe to standard input file of the process; we will not be able to read from the pipe object after this call.

status = Pipe. wait()
Closes both intput and output pipes; waits until the process finishes; the exit status of the process is returned by wait.

If pipe object is discarded, because it is no longer referenced, then the process will be stopped with signal SIGKILL.

---

kill(~pid 12345 ~signal 9)

Sends signal 9 to process identified by process id 12345

Time related functions

---

tm = time()

Tm is now the current time, the number of seconds that have passed since seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time, without including leap seconds.

[secs, minutes, hour, dayofmonth, month, year, dayofweek, isdaylightsaving] = localtime()
[secs, minutes, hour, dayofmonth, month, year, dayofweek, isdaylightsaving] = localtime(~epoch tm)

If the function is called without parameters, then the current time is broken down into component values, relative to the current timezone. These components values are returned in an array.

The function may receive an argument ~epoch – the time in epoch seconds, this is a time value returned by the time function

---

[secs, minutes, hour, dayofmonth, month, year, dayofweek, isdaylightsaving] = gmtime()
[secs, minutes, hour, dayofmonth, month, year, dayofweek, isdaylightsaving] = gmtime(~epoch tm)

If the function is called without parameters, then the current time is broken down into component values, relative to the UTC timezone (universal time zone). These components values are returned in an array.

The function may receive an argument ~epoch – the time in epoch seconds, this is a time value returned by the time function

---

sleep(~delay 3)
The program does nothing for three seconds

sleep(~delay 3 ~units 1)
The program does nothing for three milliseconds

sleep(~delay 3 ~units 2)
The program does nothing for three microseconds

Functions for green threads (threads are also known as pooh routines)

---

makethread

Some explanation of the concept of green threads, which are also known as pooh routines.

When a regular function is called, the statements that make up the body of the function are evaluated, when the function is completed, the computer magically return to the spot from where the function was called and then continues to evaluate from right after the function call.

Threads are different: A thread is a unit that call its own functions, as usual. Each such thread performs a task of its own. Now lets imaging two threads. The consumer of honey thread – also known as the Bear thread, and the producer of honey thread the Bee thread.

Imagine the consumer thread, the Bear thread; the Bear is always busy with playing, humming and visiting Christopher robin or his many friends. From time to time the Bear thread is hungry. The bear thread calls the **resume** function on the bee thread, the Bear thread goes to sleep until it gets the data back from the Bee thread.

The bee thread is resumed; Imaging the producer thread of bees; the bees are always busy in a loop with sending scout bees to find new flowers, then send out the worker bees to gather the nectar and bring it back to the he beehive; the worker bees have to stack up the honey somewhere. From time to time some new honey is ready; the problem is that you can't just stop the bee thread and return to a calling function, as if it were a regular function, that would mess up the state of the bees. So the bee thread passes the honey data to the function **yield**, which suspends the bee thread, passes the data back to the calling Bear thread,

Now the Bear thread is once again active, the **resumethread** function returns with the data it received from the bee thread (the honey if you have noticed).

Note that only one process is active at a given time; the other threads are suspended.

Also with this example we all know now why green threads are known as Pooh routines.

```
th := makethread ( ~func myrange )

sub myrange( from, to )
   i = from
   while i <= to
     threadyield0( ~yieldval i )
      i = i + 1
   end
end
thread( ~from 1 ~to 10)
```

The function makethread takes a regular function and returns a function reference marked as thread. A new thread is created when calling the function via returned reference.

[isrunning, yieldvalue] = threadyieldvalue(~thread th)

for the user of a thread: this function returns the first yield value of the thread th; Note that a thread is started as if it where a function, so the yield value cannot be returned as thread function return value, hence this function.

[isrunning, value] = resumethread(~thread th)
[isrunning, value] = resumethread(~thread th ~message msgtothread)

Given that thread th (argument **~thread**) is suspended; resumes the thread th and suspends the current thread. The current thread is resumed again, when the thread th calls threadyield0 or threadyield; the return value of resumethread is an array with two elements: the first one is a status **isrunning** is 1 if thread th can be resumed (i.e. it has not stopped). The second value is the yield value of thread th; the value that th has passed to threadyield0 or threadyield.

[requestexit, msgtothread] = threadyield(~yieldval value)
[requestexit, msgtothread] = threadyield( )

Assumes that the current thread is a pooh thread; that means it has been started via call to function reference returned by makethread.

The function threadyield suspends the current thread, and passes control back to the thread that started it, the argument value value of parameter ~yieldval is passed back to the calling thread. If the current thread has just been started, then the calling thread can get this value from return value of function threadyieldvalue ; if we have been activated via call to resumethread then return value of resumethread will be the yield value. The parameter ~yieldval is optional; if it is not given then Null value is passed to the calling thread as the yield value.

Return values: the function threadyield returns an array with two values.

- The first value requestexit; if this value is not zero (true) then the calling thread has just called stopthread ; this means that our thread is supposed to clean up any resources that it owns and exit. Owned resources that need clean up are files - for example.
- The second value msgthread; this is the value that resumethread function passes as ~message parameter. Null is returned if the function did omit the ~message parameter.

---

msgtothread = threadyield0(~yieldval value)
msgtothread = threadyield0( )

Assumes that the current thread is a pooh thread; that means it has been started via call to function reference returned by makethread.

The function threadyield0 suspends the current thread, and passes control back to the thread that started it, the argument value value of parameter ~yieldval is passed back to the calling thread. If the current thread has just been started, then the calling thread can get this value from return value of function threadyieldvalue ; if we have been activated via call to resumethread then return value of resumethread will be the yield value. The parameter ~yieldval is optional; if it is not given then Null value is passed to the calling thread as the yield value.

The function threadyield0 returns the value that resumethread function passes as ~message parameter. Null is returned if the function did omit the ~message parameter.

If the current thread (thread th) is resumed, and the calling thread requests to stop the thread by calling stopthread, then this thread is finished. So this function is suitable if thread th does not hold any resources that need to be cleaned up; resources like files need to be closed for example.

---

threadexit()

A running thread calls this function to finish and exit. Equivalent to returning from the threads main function.

---

stopthread(~thread th)

stops a running thread; if th is a function reference to a thread (marked by makethread) and the thread is running. The current thread is suspended and thread th is resumed;

- if thread th has been suspended due to calling function threadyield, then function threadyield returns status 0; that means the thread must clean up any owned resources and exit.
- If thread th has been suspended due to calling function threadyield0, then the thread finishes; this is suitable if thread th does not hold any resources that need to be cleaned up; resources like files need to be closed for example.

---

isthreadmain()

A thread function returns 1 if it has been started as thread; i.e. the function has been marked by makethread and then called via return value of makethread. This is useful if you want to write a function that is a thread on the one hand, and behaves as a regular function when called as a regular function. For example the range function returns an array when called as a regular function, but is a thread when used from a for loop.