Open Object Rexx 5.1
Classic Short Reference

Jochem Peelen

14 Apr 2025

The popular IBM *REXX Reference Cards* always were a bit too brief for me. So I wrote a Short Reference for myself. Over time, this eventually developed into the current document. It provides 99 % of the information I need to look up when working with ooRexx. I hope it is useful for others, particularly those who also code classic style.

Purpose

- The document applies to Windows 7 (or later) with ooRexx 5.1 installed.
- Its first part is a **compact** syntax presentation of what I see as frequently used language elements, grouped by type:
  - character strings,
  - strings of words,
  - program loops,
  - arithmetic (including Rosettacode library **RXM**),
  - time and date,
  - managing files and directories,
  - reading and writing of files (the „new“ and the „conventional“ way),
  - bits and bytes (hexadecimal, decimal, binary presentation).
- The second part, starting on page 35, is an introduction for newcomers how to use the following „multitools“ ooRexx provides:
  - stem variables,
  - arrays –plus capabilities of method **SORT2**– (mostly written with migration from stems to arrays in mind),
  - **USE ARG** –including **::ROUTINE**– for subroutines and function libraries.
- The third part, starting on page 51, shows less frequently referenced but important matters:
  - „Classic“ versus „object oriented“ way to use **~sortwith** for solving a simple sort problem,
  - some ooRexx fundamentals,
  - how to read the syntax diagrams of this document,
  - alphabetical index,
  - table of contents (immediately accessible on the back page).
- **Potential ooRexx users** could use this document to get a feeling of the language. I particularly recommend the chapters on fundamentals (page 55) and how numbers are processed (page 16) for this.

Hint: If in doubt about how a function or method actually reacts to certain inputs, program **rexxtry** offers the ability to interactively find out. It comes with ooRexx.

This document reflects my own experiences with ooRexx 5.0 and 5.1 running on Windows 7, 8.1 and 10. The latest build I installed was r12924 dated 21 Nov 2024. This reference was created in my spare time. Be prepared to encounter typos and factual errors.

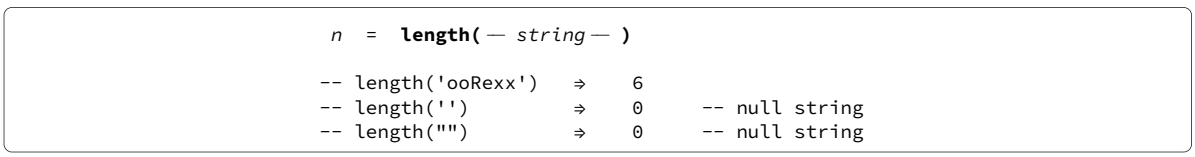
From many details it will become obvious that English is not my native language. The original German version is titled **Open Object Rexx 5.1 Kurzreferenz für Klassiker**. This English language edition was created for the *36th International Rexx Language Symposium 2025* in Vienna, Austria.

Text set with pdfLaTeX in URW Palladio; diagram font: Adobe SourceCodePro

1 Character Strings

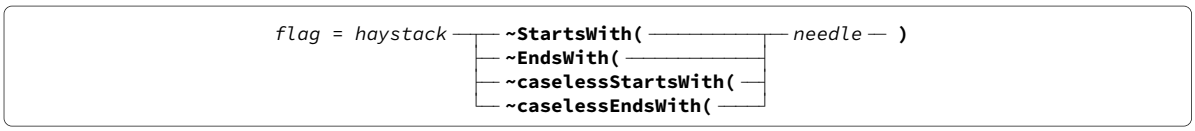
1.1 Informations About Strings

Length

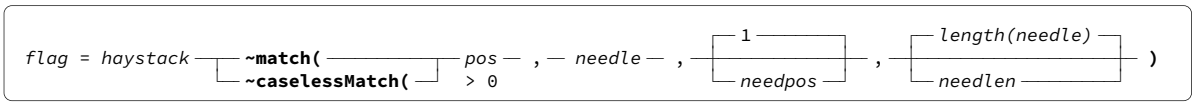


Returns the length of *string*, which is 0 in case of a null string.

Testing Strings (True/False)

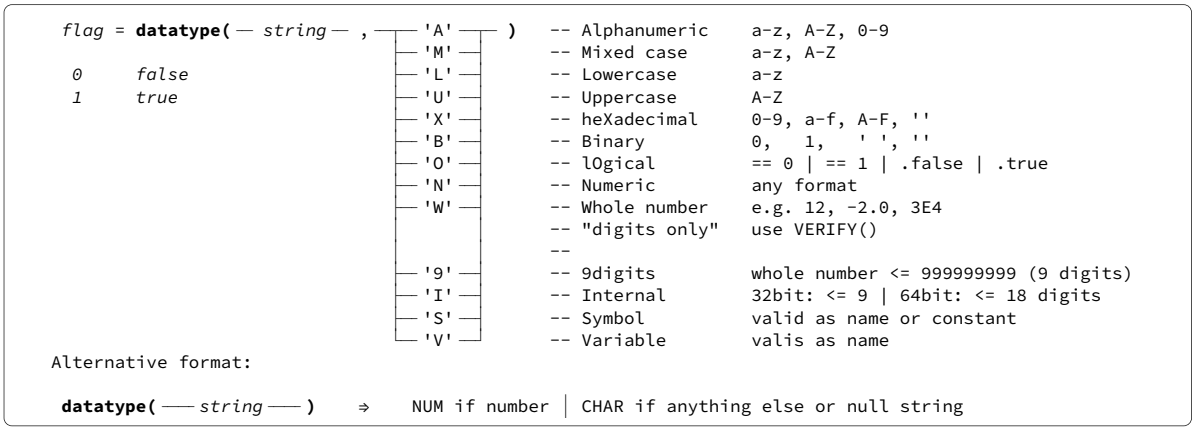


Returns 1 if string *haystack* starts/ends with *needle* or if both are equal. Returns 0 otherwise.[1]



Returns 1 if string *needle* starts in column *pos* of *haystack*, else 0. By using *needpos* and *needlen*, only a substring of *needle* (!) will be compared.

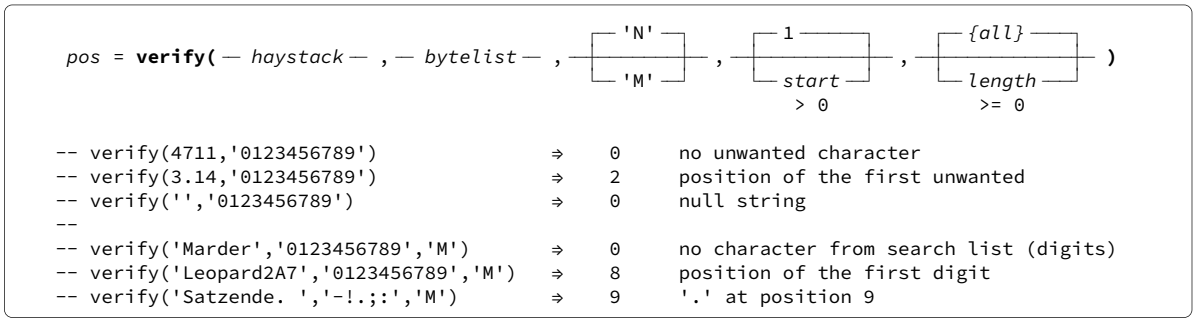
Returns 1 if the character at column *pos* of *string* is also in *bytelist*, else 0. A 0 will also be returned if *string* or *bytelist* are null strings or if *pos* is beyond the length of *string*.



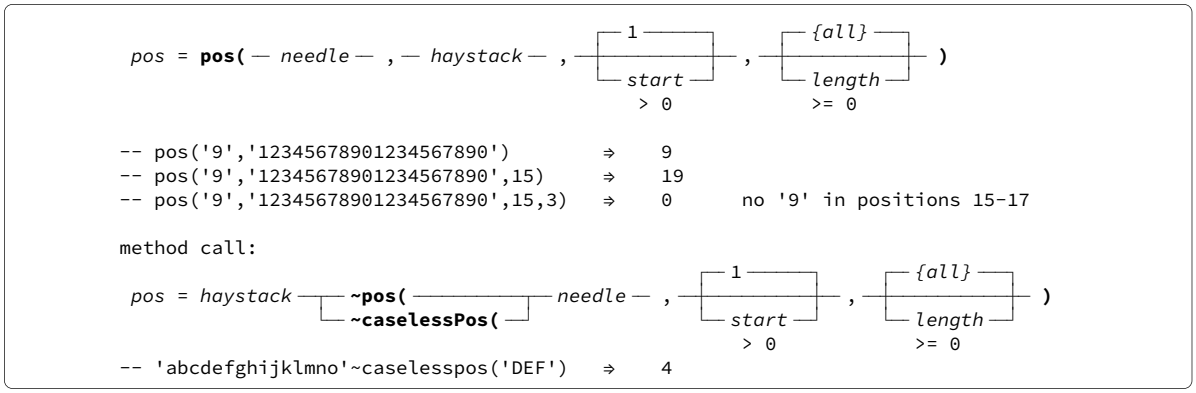
Returns 1 if *string* is of the data type indicated by the letter, else 0. If the alternative format without type letter is used, NUM will be returned if *string* is numeric, else CHAR.

[1] **Caseless** does not recognize foreign language umlauts or diacritical symbols as letters.

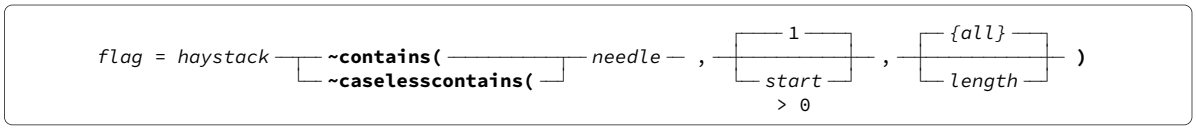
Searching in Strings



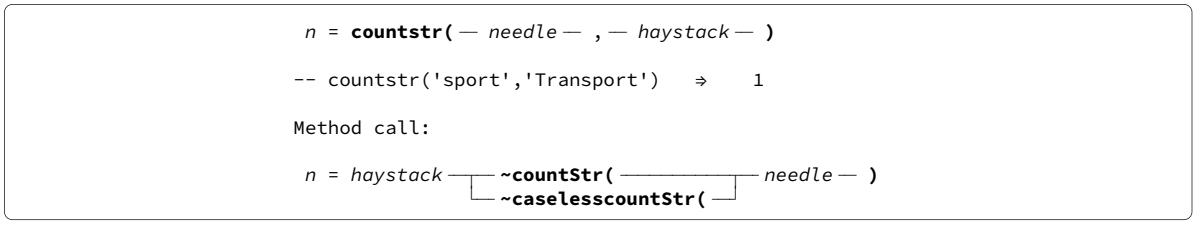
In **N**[omatch] mode, *bytelist* acts as a list of *valid* characters. A 0 for „none invalid“ is returned if all characters of *haystack* are also in *bytelist* – or if *haystack* is a null string. Otherwise the position of the first invalid character in *haystack* is returned.
In **M**[atch] mode, *bytelist* acts as a list of *unwanted* characters. A 0 for „none unwanted“ is returned if no character from *bytelist* is present in *haystack*. Else position of the first unwanted character in *haystack* is returned.
Use *start* and *length* to limit the comparison to a substring of *haystack*. Length 0 always returns 0.



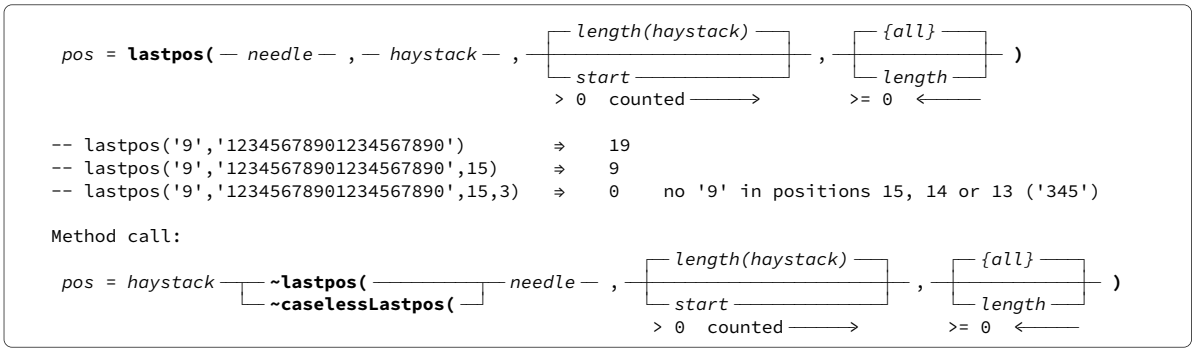
Returns the position of the first character of *needle* in *haystack* or 0 if *needle* is not found.



Works like **pos()**, but returns either 1 or 0, as used in conditional terms (page 56).

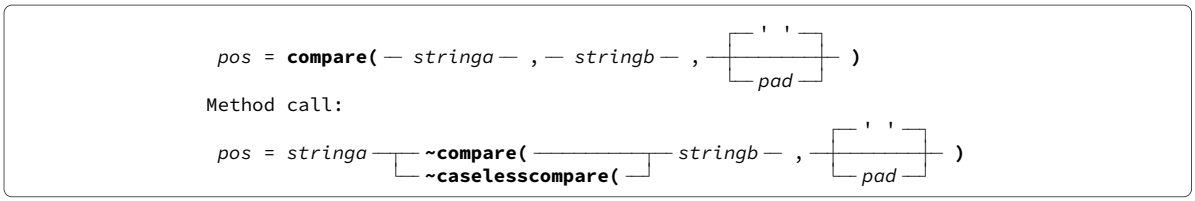


Counts how often *needle* is completely contained in *haystack*.

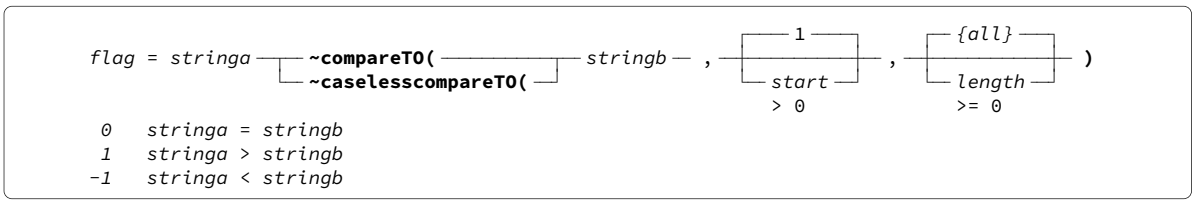


Begins the search for *needle* at the last character of *haystack* and scans toward the first character (←). The returned position (first character of *needle*) is counted as usual (→). If *needle* is not found, or either string is a null string, 0 is returned.
start (→) begins the scan at any other position of *haystack*.
length specifies how many characters of *haystack* are scanned (←). 0 is allowed and will return 0.

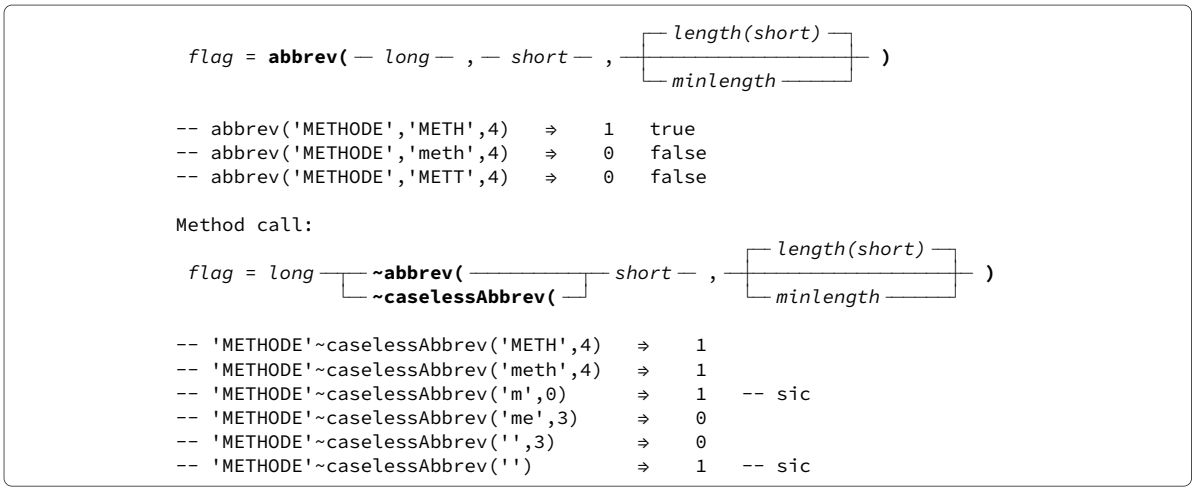
Comparing Strings



Returns 0 if both strings are equal. Else the position of the first not matching character is returned. If one string is shorter, it will be padded with character *pad* before the comparison.

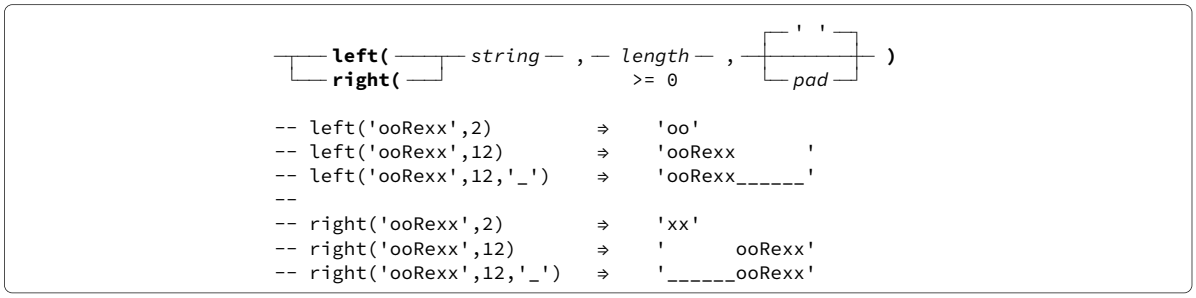


By returning 0, 1, or -1 it is reported if the strings are equal or which is the larger of the two, based on the sorting sequence for character strings. No padding is done.
Use *start* and *length* to limit the comparison to a substring of the input. Length 0 will always return 0.



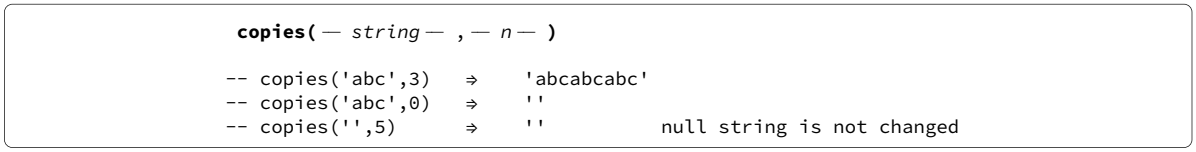
Returns 1 if *long* starts with the characters of *short*, otherwise 0. A minimum length of *short* may be required.

1.2 Truncation and Padding



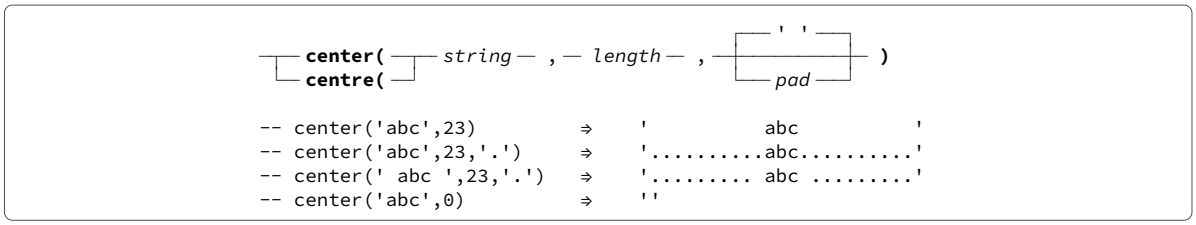

```
-- left('ooRexx',2)         ⇒    'oo'
-- left('ooRexx',12)        ⇒    'ooRexx      '
-- left('ooRexx',12,'_')    ⇒    'ooRexx______'
--
-- right('ooRexx',2)        ⇒    'xx'
-- right('ooRexx',12)       ⇒    '      ooRexx'
-- right('ooRexx',12,'_')   ⇒    '______ooRexx'
```

Returns a string of *length*, by truncating or padding with *pad* as required. **Left** makes the first character of *string* the leftmost character of the result. **Right** makes the last character of *string* the rightmost character of the result and, if necessary, truncates on the left.




```
-- copies('abc',3)   ⇒    'abcabcabc'
-- copies('abc',0)   ⇒    ''
-- copies('',5)      ⇒    ''          null string is not changed
```

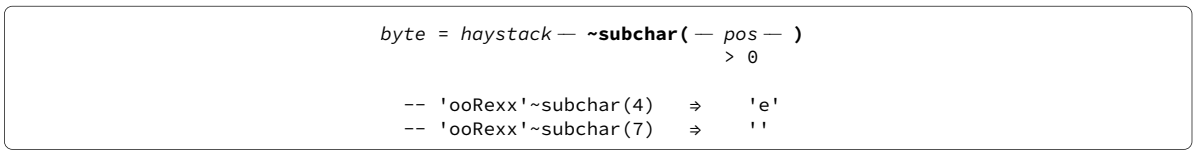
Returns *n* copies of *string*.




```
-- center('abc',23)         ⇒    '          abc          '
-- center('abc',23,'.')     ⇒    '..........abc..........'
-- center(' abc ',23,'.')   ⇒    '......... abc .........'
-- center('abc',0)          ⇒    ''
```

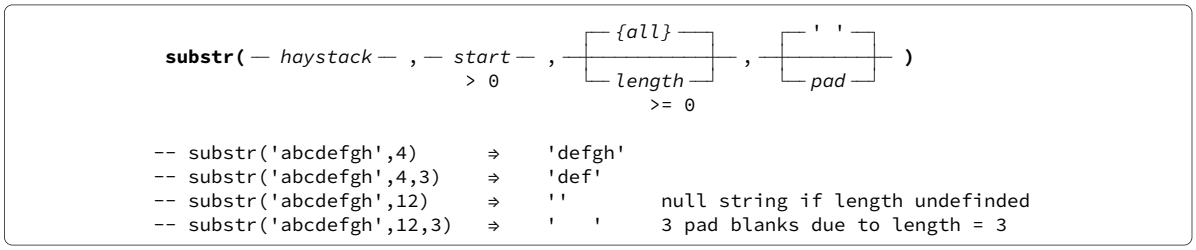
Returns a string of length *länge* with *string* centered in it. The result is truncated or padded with *pad* as necessary. If an odd number of characters has to be removed or added, the right half has one character more truncated or added.

1.3 Read Parts of Strings




```
-- 'ooRexx'~subchar(4)   ⇒    'e'
-- 'ooRexx'~subchar(7)   ⇒    ''
```

Returns the character from position *pos* in *haystack* or the null string if *pos* is beyond the end of *haystack*.




```
-- substr('abcdefgh',4)      ⇒    'defgh'
-- substr('abcdefgh',4,3)    ⇒    'def'
-- substr('abcdefgh',12)     ⇒    ''        null string if length undefinded
-- substr('abcdefgh',12,3)   ⇒    '   '     3 pad blanks due to length = 3
```

Returns the part of *haystack* that starts at position *start* and is *length* characters long. If *length* goes beyond the end of *haystack*, character *pad* is used for padding.

Digression: Keyword PARSE

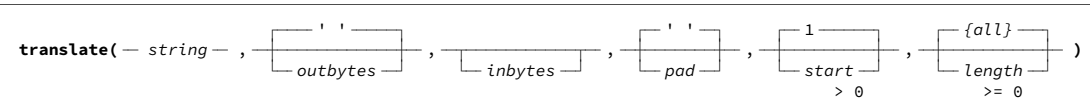
The following template is just intended to jog the memory. The large number of possibilities offered by the **parse** keyword are described in chapter 9 of the *ooRexx Reference*. An important feature is its ability of multiple –including overlapping– assignments in a single step.

```
-- Reminder of format parse var  ... for data in variables:

--          ....+....1....+....2..
example  = '   ABCD EFGH    IJKL '                  -- data in variable example

 parse var  example 4 name1 13 19 name2 21          -- by position

say '>'name1'< >'name2'<'    ⇒      >ABCD EFGH< >ei<

 parse var  example . word2 word3 .                 -- by word (note closing dot)

say '>'word2'< >'word3'<'    ⇒      >EFGH< >IJKL<

 parse var  example 9 name1 . 17 name2              -- by position and word combined

say '>'name1'< >'name2'<'    ⇒      >EFGH< >IJKL<

 parse var  example 'CD ' name2 ' '                 -- separated by quoted strings

say '>'name2'<'              ⇒      >EFGH<

separ1 = 'CD '
separ2 = ' '
 parse var  example (separ1) name2 (separ2)         -- separators in variables

say '>'name2'<'              ⇒      >EFGH<

col1 = 4
col2 = 13
 parse var  example =(col1) name3 =(col2)           -- positions in variables

say '>'name3'<'              ⇒      >ABCD EFGH<
```

```
-- Format parse value  ...  with  ... for character strings, function calls
-- and array item notation:

 parse value ─┬─ '   eins zwei    drei ' ─┬─ with ─ parsing template as above
              ├─ function(arg1,...,argn) ─┤
              └─ arrayname[n] ────────────┘          -- new: array item
```

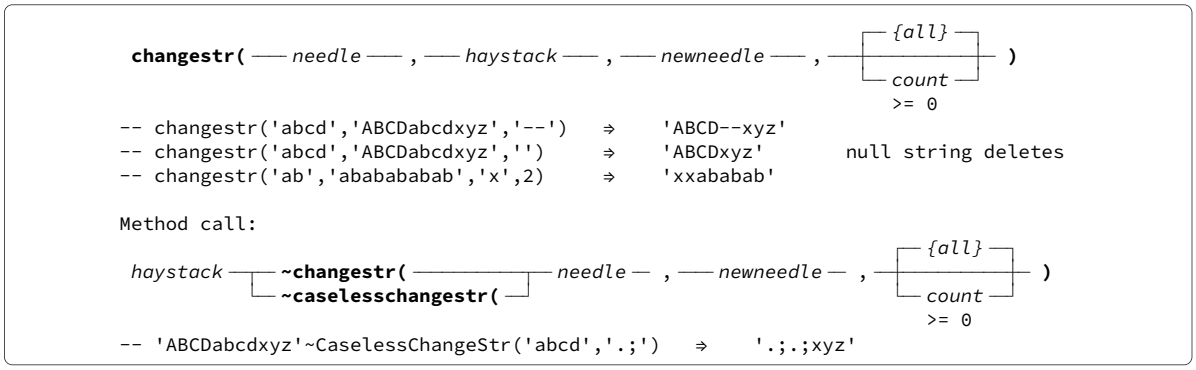
1.4 Changing Data in Strings

Changing or Deleting Characters

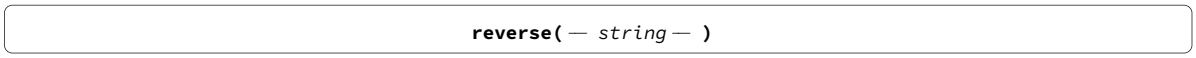



```
-- translate('1,000,000.00',',.','.,')       ⇒    '1.000.000,00'
-- translate('FöHNäRGER','ÄÖÜ','äöü')        ⇒    'FÖHNÄRGER'
-- translate('--+--*+--','#','+',,1,5)       ⇒    '--#--*+--'     -- 2nd '+' is beyond right limit
-- translate('beliebig')                     ⇒    'BELIEBIG'      -- a single argument is folded UPPER
```

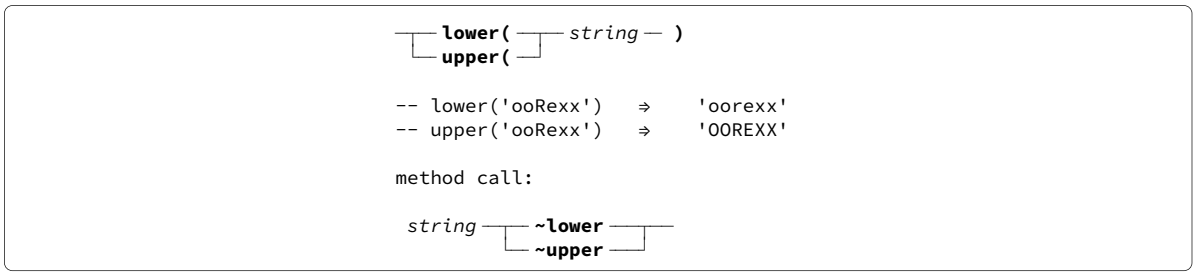
Returns *string* with the characters of list *inbytes* replaced. The replacement is taken from the identical position in list *outbytes*. By using *start* and *length* the translation can be limited to a substring of *string*. If *inbytes* is a null string or blank, no translation occurs. Should *inbytes* be missing, all characters in *string* are replaced by blanks.
If *outbytes* is shorter than *inbytes* it will be padded with *pad*. Should *outbytes* be a null string, all characters defined by *inbytes* are translated to the *pad* character.
If *string* is the only argument, letters a...z are are folded to uppercase.



Returns *haystack* with all occurrences of string *needle* changed to *newneedle*, which can be shorter or longer. If *newneedle* is a null string, all occurrences of *needle* are effectively deleted. Using *count* can limit the changes to the first occurrences of *needle*.

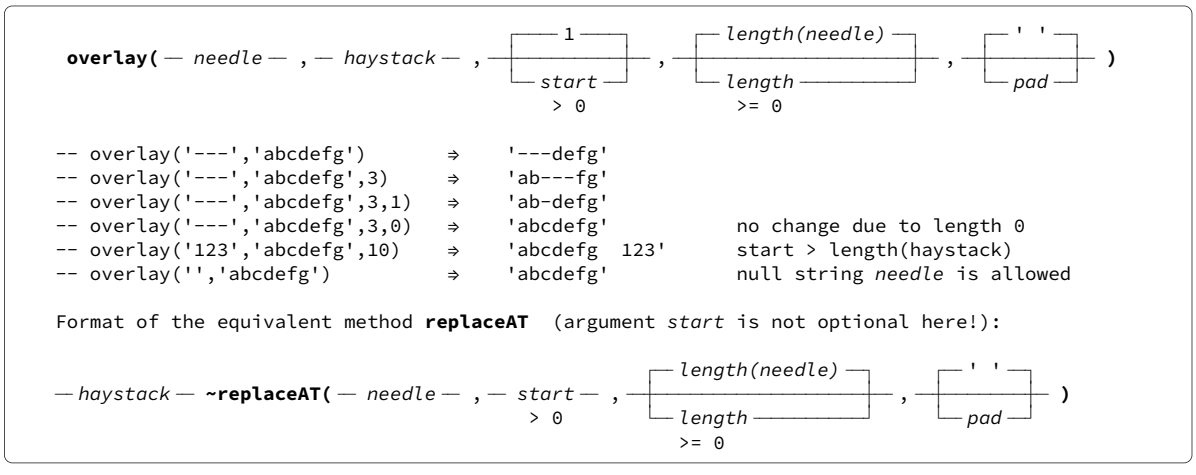


Returns the characters of *string* in reverse order *gnirts*.

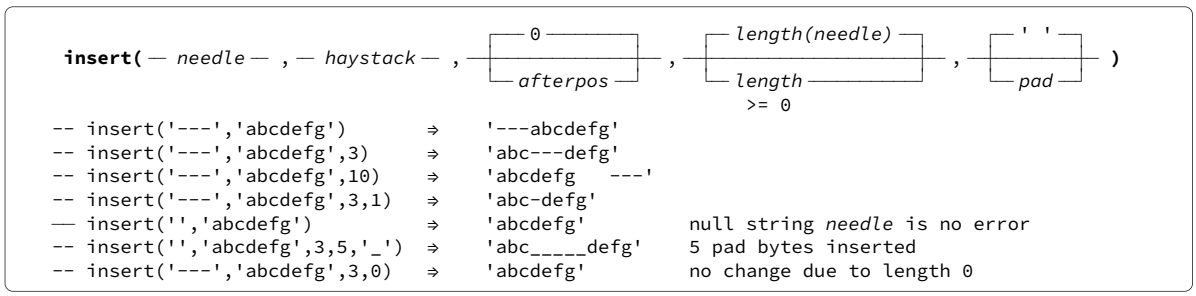


This is self-explaining. Remember that only the 26 ordinary letters are dealt with.

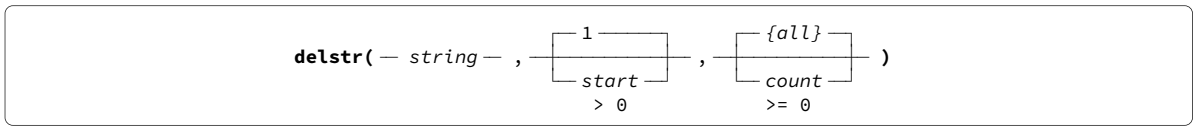
Overwriting, Inserting or Deleting Parts



Returns *haystack* overwritten from position *start* with string *needle*. If *needle* is a null string, nothing is changed. If *length* exceeds *needle*, padding with *pad* is applied, as is the case if *start* is beyond the length of *haystack*.

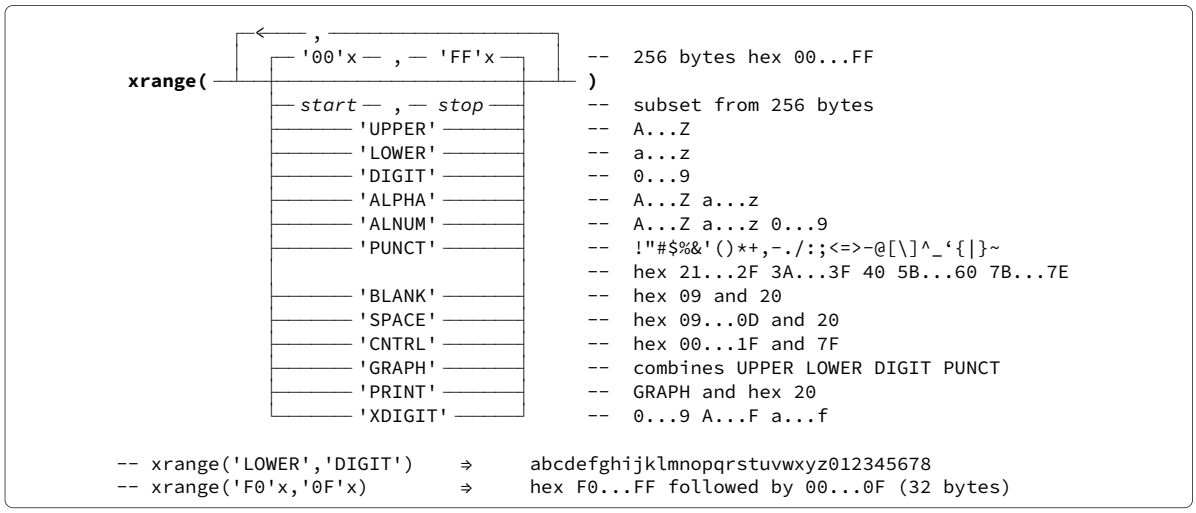


Returns *haystack* with string *needle* inserted, starting in the position following *afterpos*. Existing characters to the right of *afterpos* are shifted accordingly.



Returns *string* from which *count* characters are deleted, starting at position *start*. Remaining characters to the right of the deleted substring are shifted to the left accordingly.

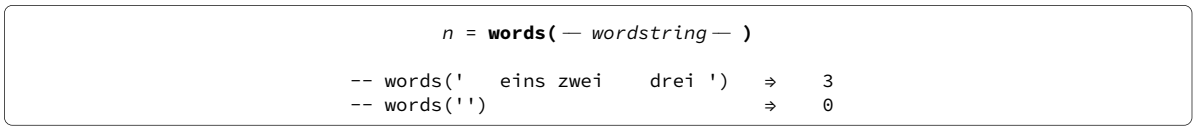
1.5 Character Types



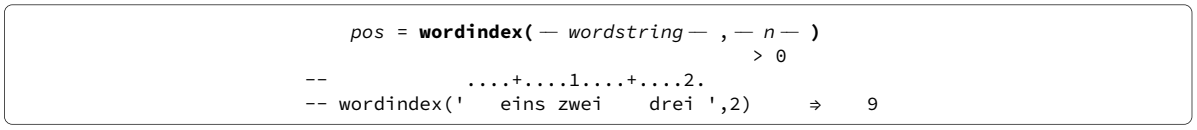
Returns a string containing all characters of the requested type. If more than one type argument is given, the result is a concatenated, continuous string. If two single characters are provided and *start* is larger than *stop*, the result is a concatenation of the bytes from *start* to hex `FF`, immediately followed by the bytes from hex `00` to *stop*. The default is hex 00...FF.

2 Word Strings

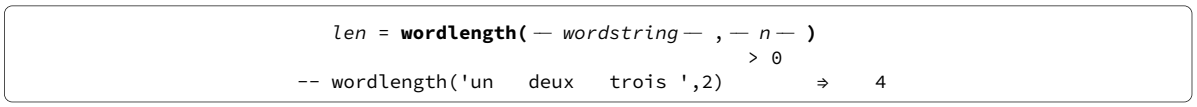
In word strings, the words are separated by **blanks** (hex 20) or **tabulator** characters (hex 09). All other characters or strings of characters are „words“. The word string may have leading and trailing blanks/tabs.



Returns the number of words in *wordstring* or 0 for a null string.

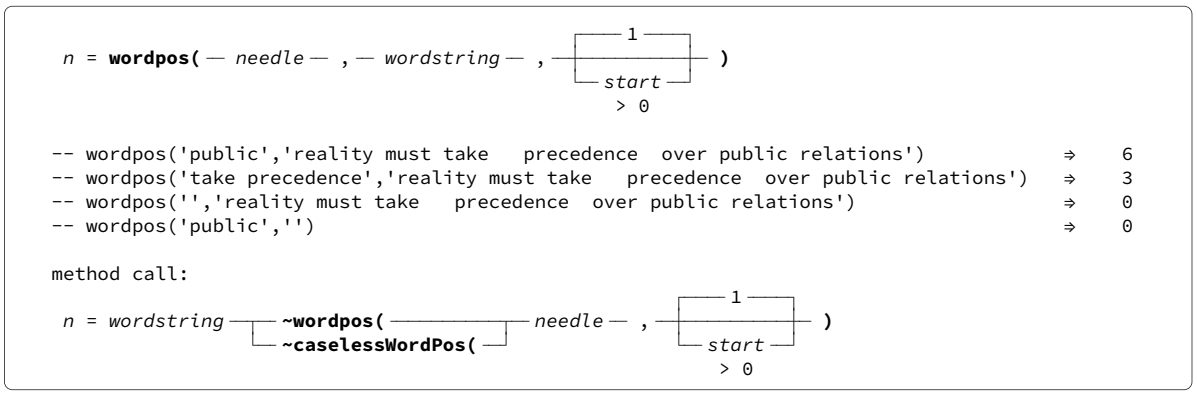


Returns the character position within *wordstring* at which the *n*-th word begins.

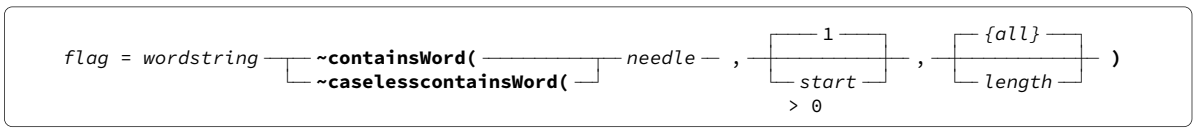


Returns the character position where the *n*-th Word in *wordstring* starts.

2.1 Searching in Word Strings

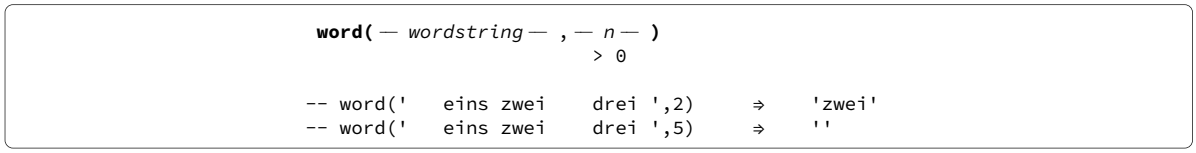


Searches in *wordstring* for *needle*, which itself may be one or more words. Returns 0 if not found. Else the character position is returned, where *needle* starts within *wordstring*. If used, *start* is the number of the word where the search begins.
0 is always returned if *needle* and/or *wordstring* are null strings or if *start* exceeds the number of words in *wordstring*.

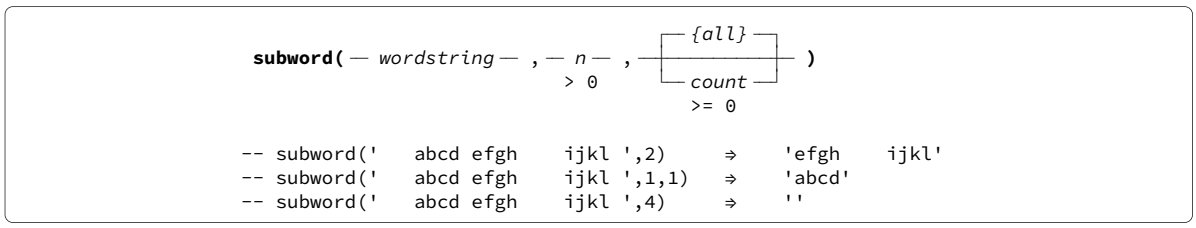


Works like **wordpos()**, but returns either 1 or 0, as used in conditional terms (page 56).

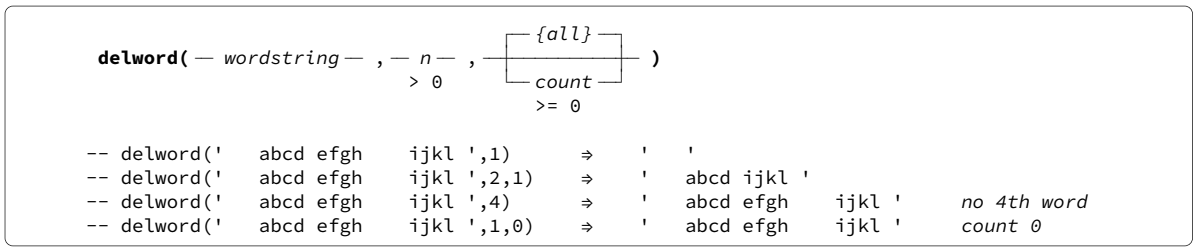
2.2 Read or Delete Parts of Word Strings



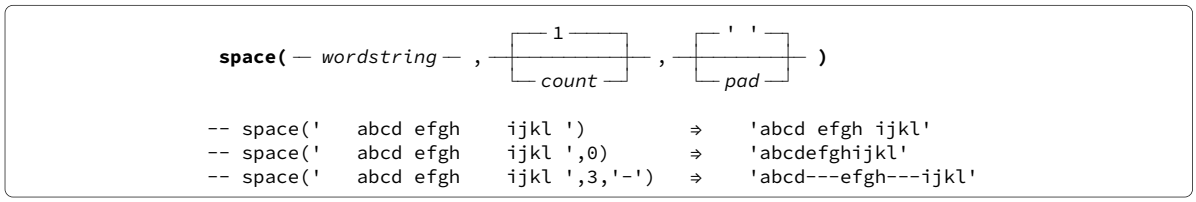
Returns *n*-th word from *wordstring*.



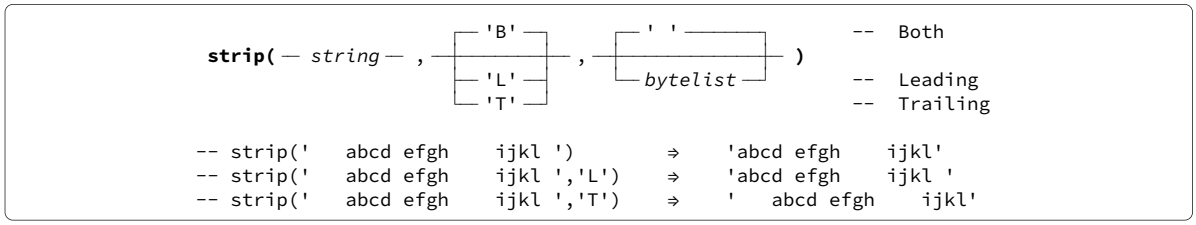
Returns *count* words from *wordstring*, starting with the *n*-th word.



Returns *wordstring* after removal of *count* words, starting with the *n*-th word. With each deleted word, its **trailing blanks** are also deleted.



Returns *wordstring* padded with *count* characters *pad*. Instead of blanks, any character may be used. If *count* is less than an existing gap, characters are deleted accordingly. Function **space()** changes the gaps **between** words; see also **strip()**.



Returns *string* after removal of **L**eading, **T**railing) or (default) **B**oth blanks. Using *bytelist*, multiple characters to be deleted can be supplied. If *bytelist* is a null string, nothing is deleted. Function **strip()** changes the characters **at both ends** of a word string or character string; see also **space()**.

3 Program Loops

Keyword **do** is used in ooRexx for simple **do ... end** blocks as well as loops. For the latter, additional keywords have to follow on the line after **do**. Since ooRexx 3.2 loops can also be defined using **loop ... end**.

3.1 Simple Loops

```
-- "endless" loops may be ended with keyword leave

 do forever                  |    equivalent:    loop
 ...                         |                   ...
 if ... then leave           |                   if ... then leave
 ...                         |                   ...
 end                         |                   end
```

Both examples show a loop that runs indefinitely, unless one of the keywords **leave**, **while** or **until** is used as explained later. In all following examples, **loop** could be used in place of keyword **do**.

```
 do  wholenumber
 ...    >= 0
 ...
 end


-- do 25   ⇒    loops 25 times
-- n = 0
-- do n    ⇒    loop is skipped entirely
```

In this case, the number of iterations is defined right at the beginning. Value *wholenumber* must be a positive whole number, which can be the contents of a variable. It also can be a function or method call that returns a whole number. If the number is 0, the loop is skipped. Processing immediately continues at the first instruction after **end**.

3.2 Using an Iteration Variable

```
-- to and for are tested before each iteration
-- After each iteration incr is added to iter

                                          ┌─ by 1 ──────┐
 do ... iter = start ─┬──────────────────┬─┴─────────────┴─┬──────────────────┬─...
                      └─ to — stopval ─┘ └─ by — incr ─┘ └─ for — count ─┘


-- do i=1 to 5           ⇒    iterations with i=1 ... 5
-- do i=5 to 1 by -1     ⇒    iterations with i=5 ... 1
-- do i=3 to 5 by 0.25   ⇒    iterations with i=3, 3.25, 3.50 ... 5.00
-- do i=1 to 5 for 3     ⇒    iterations with i=1 ... 3; stopped by FOR
-- do i=1 to 3 for 5     ⇒    iterations with i=1 ... 3; stopped by TO
-- do i=1                ⇒    "endless" loop with i=1, 2, 3 ...
-- do i=2 by 2           ⇒    "endless" loop with i=2, 4, 6 ...
```

Before the instructions inside the loop are executed, iteration variable *iter* is initialized to numeric value *start*. It need not be a whole number and may be negative. After each iteration, the **by** value *incr*, which my be negative, is added[1] to *iter*. Like *start*, *incr* need not be a whole number and may be negative.

[1] The arithmetic rules as described on page 16 apply.

iter can be modified within the loop to change loop execution. This variable continues to be available after the loop has ended and can be used like any other variable.
to und **by** belong together. If **by** is not specified, *incr* is 1. Keyword **by** without **to** results in an infinite loop. It has to be stopped by other keywords like **leave** for example.
for is independent from the iteration variable and sets an internal counter for the number of iterations. If this is reached, the loop ends unconditionally. If *count* is represented by a variable, changing it after loop start has not effect. During the development phase of a program, **for** can be used to avoid unintended spin loops, which would otherwise require externally forcing the program to crash. Do not forget to remove such **for** keywords in time.

```
-- Keyword end may have the iteration variable name appended as label
-- (this is optional but really improves readability)

num = 8
do  x=1 to num
...
     do  y=0  for num
     ...
          do  z=1 to 3
          ...
          end z
     ...
     end y
...
end x
say x z   ⇒    9 4   -- Iteration variables survive the loop end
```

The name of the iteration variable *iter* may be appended as a label to the **end** keyword for this loop. It makes obvious to which **do** line the **end** belongs. Particularly if nested loops are used, the code is much easier to read. This name may also be appended to keywords **leave** and **iterate** (see page 15). The ooRexx interpreter reports label inconsistencies, but does not need them for correct program execution.

Controlling Loops by Comparisons

Either **while** or **until** at the end of a **do** line can be used to conditionally stop further iterations of a loop. As long as **while** is true before the start of an iteration, execution will continue. If at the end of an iteration, condition **until** is true, the loop ends.
The same comparison operators apply as shown on page 56 for **if** and **when**. Also the same logical operators & (AND), | (OR), && (XOR) and the comma (conditional AND) can be used. It may not always be obvious at first sight what the result for loop execution is:

```
-- while is tested before the pending iteration begins


 do  ...  while — condition — ...

-- do while num <= 10    ⇒    Stop when num exceeds 10
```

As long as the **while** condition is true, the loop continues.

```
-- until is tested after each iteration


 do  ...  until — condition — ...

-- do until num > 10     ⇒    Stop when num exceeds 10
```

When the **until** condition is true, the loop stops.

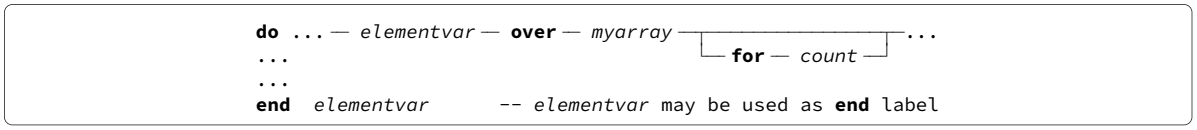
Sequence of Steps per Iteration

```
Before each iteration of the loop:
1. Leave if iteration control iter is larger than to  stopval.
2. Leave if  wholenumber is exceeded.
3. Leave if for  count is exceeded.
4. Leave if while condition is  .false

Begin iteration:
5. If keyword counter is active, increase loopnum by 1.
6. Execute the instructions in the loop.
   If iterate is encountered, skip remainder of this iteration.

After each iteration of the loop:
7. If until condition is .true  leave the loop.
8. Add by  incr to iteration control iter.
9. Continue with step 1.
```

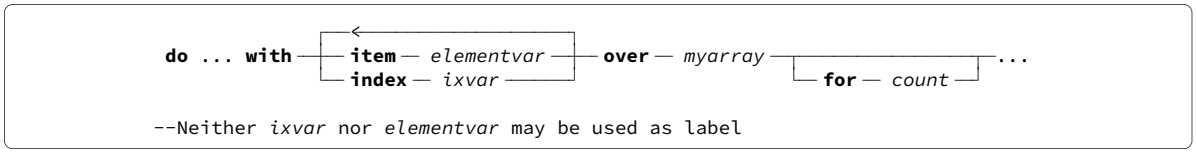
Unless step 7 has stopped the loop, step 8 –incrementing the iteration variable *iter*– is always done. After that, steps 1 to 3 before the next iteration may also stop the loop. Consequently, in most cases *iter* after the loop already has the value for the next iteration, for example value 9 after a loop with **i=1 to 8** ended. Only if the loop was stopped because of the **until** condition or the **leave** keyword, *iter* keeps the value it had during the last iteration of the loop.

3.3 New Loops for Data Collections

In ooRexx 5.1 there are 13 type of data collections defined. Two new types of loop were created in view of these collections. Welcome advantages are: it is **not necessary to know the size** of the collection. Also, **empty elements** (unused index locations amid locations with data) are allowed. They are ignored by the loops. Here, the behaviour with collection type **array** will be used as example to demonstrate how the new loop types basically work. I have no experience with the other collection types.
At the start of the loop, a logical „status snapshot“ of the data collection is taken. Changing data is possible, but remains invisible inside the still running loop.
After the loop ends, the variables *ixvar* and *elementvar* –if used– continue to exist in the program with their last value from the loop.



The loop iterates only through those index locations of *myarray* that have data (items). On each iteration, *elementvar* contains a copy of the current data element (item). Because Array is a data collection of type *Ordered*, the sequence will strictly be ascending. For other types of data collections the sequence is undefined.
If no array *myarray* exists, the name is treated as a simple variable. The loop iterates just once and *elementvar* receives the value of the variable. If no variable of this name exists, it will be the name in uppercase.
The name of variable *elementvar* can be appended as a label to the corresponding **end** keyword. Examples are shown on pages 52, 54 and 46.



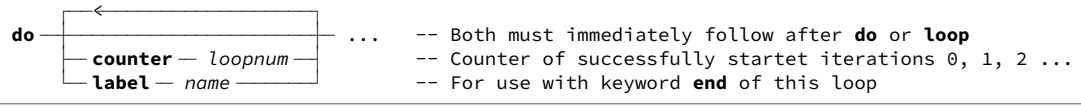
This **do...with...over** loop iterates the same way through *myarray* as **do...over** above does. If only keyword **item** is used, the result is the same as above. In addition, or alone, keyword **index** can be used to put the current array index location in variable *ixvar*. Not all collection types have an index.
With this loop variant, neither the name of *elementvar* nor *ixvar* can be used as **end** label.

```
do with item elementvar over myarray
    say elementvar
end
```

This example shows coding of a **do...with...over** loop to produce the same result as a **do...over** loop when processing an array.

3.4 Additional Control Keywords

These apply to all types of loops.



If used, these two keywords must appear immediately following keyword **do** but may be used in any sequence.
counter initializes variable *loopnum* to 0. Any successfully started iteration will increment it by exactly 1. Ending an iteration with keywords **leave** or **iterate** will not change *loopnum*. Variable *loopnum* can be read by the program, but any change will be overwritten by ooRexx at the beginning of the next iteration. This is different from the iteration variable, where ooRexx accepts and recognizes changes by the program code.
Keyword **label** defines a *name* that may as label of the corresponding **end** keyword. It makes this readability feature available in loops which otherwise do not provide a means to identify which **do...end** keywords belong together.

Keywords LEAVE and ITERATE

Keyword **leave** immediately stops the current iteration and end the loop. Processing continues with the line after the **end** keyword of the current loop.
Keyword **iterate** stops the current iteration and skips its remaining instructions. If an **until** condition is defined and true, the loop ends. Else the next iteration is started.

Nested Loops

If loops are nested and the **end** keywords have a *name* label (*name* of iteration variable, or with keyword **label**), it is possible to control an outer loop from within an inner loop. This works across multiple nesting levels.
Instruction **iterate** *name*, when used in an inner loop, will terminate it and result in an **iterate** action of the outer loop. The affected outer loop is identified by its **end** *name* statement.
Instruction **leave** *name* works the same way, but *leaves* the outer loop.

4 Arithmetic

On an ordinary Lenovo E15G4 Notebook, ooRexx needs 0.30 seconds for calculating a 1000 m long trajectory, using a fourth order Runge-Kutta-Nyström numerical integration. This is over 7 times faster than the 2.15 seconds time of flight in reality. For an interpreted language, ooRexx computes quite fast, particularly on current hardware.
If required, ooRexx can compute with a precision which is only limited by available storage (RAM). As an example, let us display the number 2 to the power of 256. Using the default precision of **9** significant, digits the result is displayed as:

```
say 2**256   ⇒    1.15792089E+77
```

To raise precision to 80 digits, the instruction is:

```
numeric digits 80
```

and all 78 digits of the result are displayed:

```
say 2**256   ⇒    115792089237316195423570985008687907853269984665640564039457584007913129639936
```

This ability is unusual for a general programming language. In addition, apart from the mathematical library **rxmath** (16 digits) that comes with ooRexx, an external library **rxm** is available.[1] It offers **nearly unlimited precision** for logarithm, exponential and trigonometric functions. For experimenting with the effects of increased precision, this comes in handy.

4.1 How Rexx Processes Numbers

The floating point arithmetic of IEEE 754, which has its origins in classic Rexx, is used with a default precision of 9 digits. In practice, this means:

- Each number ooRexx encounters is saved internally as a character string.
- If an arithmetic operation is encountered, the character string will be converted to a floating point number according to IEEE 754 and rounded to 9 significant digits. For example:

  ```
  1.1234512345     ⇒   1.12345123
  3210.1234512345  ⇒   3210.12345
  ```

- The result of the operation will also be rounded to 9 digits and then converted back to a character string.[2]
- If more places than the current digits setting are needed before the decimal point, ooRexx uses exponential form for this number. Function **format()** allows to convert any number to exponential form.

At any time in the program, keyword `numeric digits` allows to change the precision.

[1] **rxm.cls** can be be downloaded from `rosettacode.org` free of charge; see page 19.
[2] ooRexx tries to save both formats internally to reduce conversions.

4.2 Operations and Functions

The following examples use the `numeric digits 9` default, unless noted otherwise.

```
5 + 3          ⇒     8                    -- addition
5 - 3          ⇒     2                    -- subtraction
5 * 3          ⇒     15                   -- multiplication
5 / 3          ⇒     1.66666667           -- division
5 / 3          ⇒     1.666666666666667    -- division after numeric digits 16

5 // 3         ⇒     2                    -- remainder, but see modulo
12.8 // 2.5    ⇒     0.3
5 %  3         ⇒     1                    -- whole number quotient
12.8 % 2.5     ⇒     5
5 ** 3         ⇒     125                  -- whole number exponent or zero
5 ** -3        ⇒     0.008

5**20          ⇒     9.53674316E+13       -- exponential form
5**-20         ⇒     1.048576E-14
5E2 + 0        ⇒     500                  -- same as  5 * 10**2
-5E2 + 0       ⇒     -500
```

```
result = dividend — ~modulo( — divisor — )

  5~modulo(3)      ⇒    2
  (-5)~modulo(3)   ⇒    1
  -5~modulo(3)     ⇒   -2   --  wrong  because interpreted as -(5~modulo(3))
  number = -5
  number~modulo(3) ⇒    1
```

In ooRexx, the tilde character has priority over the minus sign. If used in the code, negative numbers need to be put in parentheses to force correct interpretation of the expression. This is not necessary if the negative number is represented by a variable or returned by a function call, because no minus appears in the code.

```
number ──┬─ ~ceiling ─┬─             -- natural number towards +infinity
         └─ ~floor ───┘              -- natural number towards -infinity

  (2.12)~ceiling     ⇒    3
  (3)~ceiling        ⇒    3
  (-2.12)~ceiling    ⇒   -2
  -2.12~ceiling      ⇒   -3    -- wrong because interpreted as -(2.12~ceiling)
  x = -2.12
  x~ceiling          ⇒   -2

  (2.12)~floor       ⇒    2
  (3)~floor          ⇒    3
  (-2.12)~floor      ⇒   -3
```

```
    digits()                        --   current numeric digits setting

    abs( — number — )               --   absolute value of number

    sign( — number — )              --   1 | 0 | -1

                ┌─← , ─────┐        --   returns from a list of numbers...
  ──┬─ max( ─┬──┴─ number ──┴─ )    --   the largest
    └─ min( ─┘                      --   the smallest
```

The list of numbers when determining the largest or smallest, is only limited by available storage.

Formatting Numbers

```
-- Read placeholder {#} as: use places as required to display number as it is

 format( — number — , ┌─{#}─────┐ , ┌─{#}─────┐ , ┌─{#}─┐ , ┌─digits()─┐ )
                      └─wholenum┘   └─decimal─┘   └─exp─┘   └─estart──┘
                          > 0           >= 0        >= 0        >= 0
                                                  0 prevents
-- format(3.5678,4,2)   ⇒    '   3.57'           exponential
-- format(-30,4,2)      ⇒    ' -30.00'           notation
-- format(30,,,,0)      ⇒    '3.0E+1'
-- format(30,,,3,0)     ⇒    '3.0E+001'
-- format(30,4,2,3,0)   ⇒    '   3.00E+001'
```

Returns *number* rounded to *decimal* decimal places. There will be *wholenum* places before the decimal point, padded with blanks. If a minus sign is possible, increase *wholenum* by one. Insufficient space is a runtime error. If not set, no padding will be done.
Setting *estart* to 0 will force exponential form, independent of the number. If not set, the default rule mentioned above (whole number part exceeds **numeric digits**) applies.
If used, *exp* sets the places for the exponent, padded with 0. Value 0 prevents exponential form, even if *estart* is 0. Insufficient space for the number is a runtime error. If not set, as many places as necessary are used for the exponent.

```
                    trunc( — number — , ┌─0─────┐ )
                                        └─decimal┘

-- trunc(3.14159)            ⇒    3
-- trunc(3.14159,4)          ⇒    3.1415
-- trunc(3.14159,7)          ⇒    3.1415900
-- trunc(12345678987.123,2)  ⇒    12345679000.00
```

Returns *number* cut off or padded with 0 to *decimal* places. If necessary, a decimal point is added. The last example shown is due to the first processing step of rounding to 9 digits before the truncation is done.

4.3 Included Library RXMATH

Part of ooRexx installation is a mathematical functions library (`rxmath.dll`) which contains C-Library functions. To use it in an ooRexx program, add after the last program line the directive:

```
::REQUIRES rxmath LIBRARY

-- This replaces the pre-version 4.0 instructions:
-- call RxFuncAdd 'MathloadFuncs','rxmath','MathLoadFuncs'
-- call MathloadFuncs
```

This in effect tells ooRexx to include library `rxmath.dll` in any search for unknown subprogram names.
When testing this library with 2726 numbers, *Walter Pachl*[3] found that in about 30 percent of the cases, the last decimal place was off by 1 from the correct result. This will probably not affect ordinary use, but should be known to users.

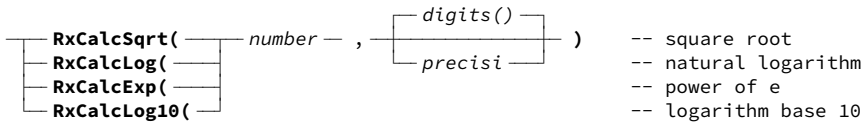
Number Pi

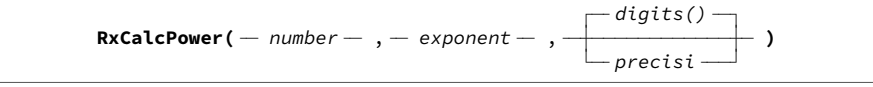
```
                    RxCalcPi( ┌─digits()─┐ )
                              └─precisi──┘

-- RxCalcPi()     ⇒    3.14159265
-- RxCalcPi(16)   ⇒    3.141592653589793
```

[3] *What's Wrong with Rexx ?* Presentation, IBM Sindelfingen 2004

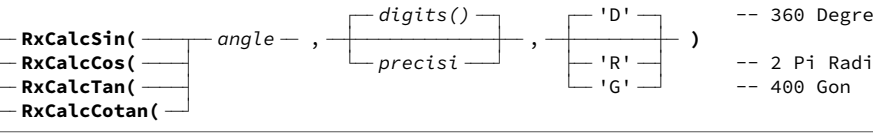
For this and all other RXMATH functions, the default is to use the current **digits()** value of the calling program for rounding the result. For any value larger than 16, the library uses 16 (*double precision* of the C-Library). Alternatively, a precision may be requested by using argument *precisi* which must be a whole number in the range 1 to 16.

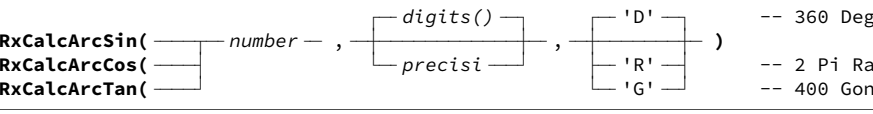
Logarithms and Powers



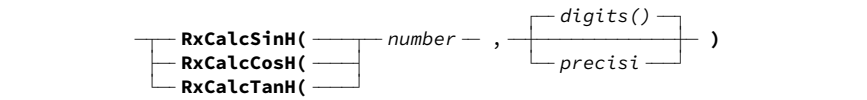


Trigonometric Functions





Hyperbolic Functions



4.4 External Library RXM

The same mathematical functions as RXMATH, but settable to much larger precision, are provided by ooRexx class file `rxm.cls` authored by *Walter Pachl*. It is programmed in ooRexx, and consequently not as fast as the compiled C library RXMATH. For the trajectory computation mentioned on page 16, with a precision of 16 digits, RXM takes 1.22 seconds instead of 0.30 seconds. If precision is set to to 32 digits, runtime grows to 2.69 seconds. RXM internally uses a precision of 10 digits more than requested (100 more for logarithm computation).
To use the RXM library, add after the last line of your program:

```
::REQUIRES rxm.cls
```

Extension `.cls` for files containing ooRexx classes is a convention, not a fixed requirement. Since February 2020 it is possible to leave it out, because **::REQUIRES** now by default searches for `.cls` files first.

REQUIRES and the ooRexx Prologue

If the target of a **::REQUIRES** directive is an ooRexx file, it is automatically searched for a „Prologue“. If it exist, its code is executed. The Prologue is formed by the code lines, if any, which **precede** the first directive in the file. A directive is an instruction beginning with two colons **::**... In the case of `rxm.cls` the prologue only has a single code line:

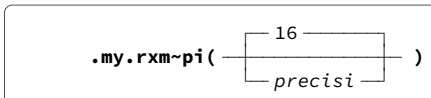
```
.local~my.rxm = .rxm~new(16,"D")  -- original prologue line
.local~my.rxm = .rxm~new(50,"R")  -- Attention: changed prologue line 2024-12-23
```

This creates a new object named `.my.rxm` which contains the attributes and methods of class RXM. At the same time it sets default precision to 50 digits and selects the 2π circle (Radian) as trigonometric unit.[4] It makes the methods of the library callable by prefixing `.my.rxm~` to it. Environment `.local` is normally active and not explicitly needed.

```
.my.rxm~precision=32
.my.rxm~type='D'
```

When used in a program, these instructions change defaults to 32 digits and 360 degrees (D). Alternatively, both parameters may be specified per method call. RXM is not able to automatically use the precision of the calling program.

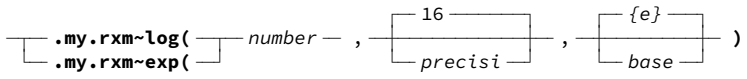
Methods in RXM



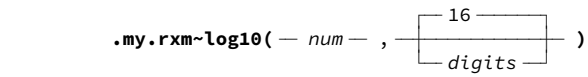

```
-- .my.rxm~pi      ⇒    3.141592653589793
-- .my.rxm~pi(64)  ⇒    3.14159265358979323846264338327950288419716939937510582097494459 2
```

The parentheses ( ) may be omitted if no argument is used.

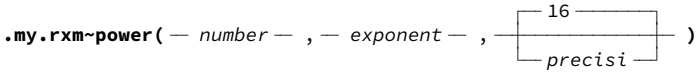
Logarithms and Powers

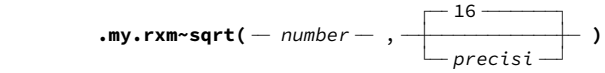



```
-- .my.rxm~log(2)       ⇒    0.6931471805599453      log base e
-- .my.rxm~log(2,,10)   ⇒    0.3010299956639812      log base 10
-- .my.rxm~log(2,,2)    ⇒    1                       log base 2
-- .my.rxm~exp(-0.5)    ⇒    0.6065306597126334      e power -0.5
```

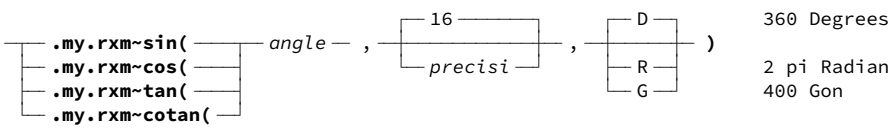


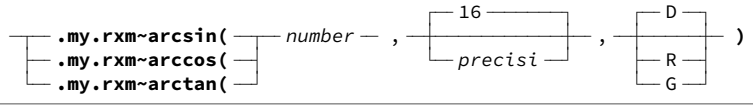

```
-- internally uses ~log
```





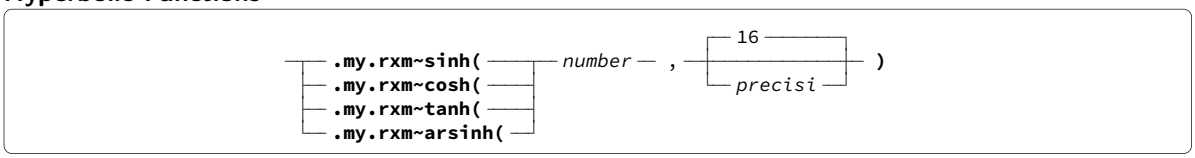
Trigonometric Functions





[4] **Attention:** Libraries `rxm.cls` downloaded before 2024-12-23, default to 16 digits and the **360 degree (D)** circle. The old defaults are still shown in the RXM method diagrams above.

Hyperbolic Functions



Compared to RXMATH, method **arsinh** is new.

Calling RXM Functions

RXM is usable with the familiar function call syntax, like RXMATH.
To do that, the name prefix **RxCalc...** has to be replaced by **rxm...**. The remainder of the name remains the same, for example:

in place of **`RxCalc`**`Sin(...)` *use* **`rxm`**`Sin(...)`

The number and order of arguments is the same. The functions do an additional syntax check before calling the corresponding method, which in this example would be `.my.rxm~Sin(...)`.
From the user view, calling RXM methods via its functions has the advantage of additional syntax checking and error messages, if required. On the other hand, computing intensive programs may be faster, if the methods are called directly. The run times mentioned on page 19 were obtained using method calls.

Downloading RXM

The library can be downloaded from `rosettacode.org` via menu **Explore**, select **Tasks** to see an alphabetic list. Go to **Trigonometric Functions** to see a list of programming languages. Then scroll to **ooRexx**. You can also use the direct link:

`rosettacode.org/wiki/Trigonometric_functions#ooRexx`

Rosettacode presents an HTML file containing descriptions and code. Use „Cut and Paste“ to save the parts in individual files as recommended below:

- The first text block, titled **rxm.cls**, is a help text and can be saved as file `rxm.txt`.
- It is followed by ooRexx code for demonstration and functional test after download of the library. Recommended is saving it as file `rxmdemo.rex`.
- The third block, headed **Output** is an output listing of running the above demo and test program. It could be saved as `rxmdemo.txt`.
- Finally, the last, very large block (1047 lines) marked **Package rxm** is the library code which should be saved as `rxm.cls`.

My recommendation is to copy file `rxm.cls` to the ooRexx installation directory, to make it accessible to all users. The ooRexx installation program will leave it alone on future ooRexx updates, just like any other local programs you may add to the Installation – as long as you avoid duplicate names.

Note on Rosettacode
`rosettacode.org/wiki/RxTriangle.rex` points to a most recently (February 2025) added library by *Walter Pachl*, which can be used for geometrical calculations: distances, crossing points, angles between lines etc.
`rosettacode.org/wiki/Category:OoRexx` takes you to a list of ooRexx programs on Rosettacode (currently more than 230). Lists are shown only in parts of up to 200 entries. You need to click on *next page* near the bottom to scroll. If `OoRexx` in the link is replaced by string `REXX`, a list of about 1100 REXX programs will appear.

5 Time and Date

The ooRexx calendar covers the range from Monday, 01 January 0001 to Friday, 31 December 9999. All days are consecutively numbered from 0 to 3652058, which is a format called *Basedate*. **Spans** between dates can be computed easily. The Gregorian calendar is being used, as if it had existed from the start. Leap seconds, for the first time inserted in 1972 and most recently in 2017, are ignored. In Windows XP, 7, 8.1 and 10 Rexx uses a clock resolution of one millisecond (3 decimal places). Any additional decimal places are always zero, as can be seen in the following examples.

```
-- Format for current time of the computer clock or stopwatch:

          ┌─ 'N' ─┐          ⇒  09:46:37
 time( ───┼───────┼─── )
          ├─ 'L' ─┤          ⇒  09:46:37.397000
          ├─ 'C' ─┤          ⇒  9:46 a.m
          ├─ 'H' ─┤          ⇒  9                  0...23        full hours since 00:00
          ├─ 'M' ─┤          ⇒  586                0...1439      dto. minutes
          ├─ 'S' ─┤          ⇒  35197              0...86399     dto. seconds
          │       │
          ├─ 'F' ─┤          ⇒  63692300797000000  microseconds since 1 Jan 0001 00:00
          ├─ 'O' ─┤          ⇒  7200000000         local microsecond delta to UTC
          ├─ 'T' ─┤          ⇒  1556668800         seconds      since 1 Jan 1970 00:00
          │       │
          ├─ 'E' ─┤          ⇒  152.114000         Elapsed: current stopwatch seconds
          └─ 'R' ─┘          ⇒  152.114000         dto. plus reset of stopwatch to 0.000000
```

Returns current time in the requested format.

Stopwatch Functionality

On its first call in a program **time(E)** as well as **time(R)** always return 0 and at the same instant start an internal time counter. Any later call to **time(E)** returns the current value of that counter, which continues running. Call **time(R)** does the same, plus resetting the counter to 0 each time.
A conventional subprogram (defined by a label in the same program file as the caller) inherits the current value of a running time counter. But the counter in the caller continues running independently of any resets by the subprogram.
To a subprogram which is defined by a **::ROUTINE** directive, the callers time counter is invisible.

Convert Time Formats

```
-- Converting the format of a time instant
-- between 00:00:00.000000 and 23:59:59:999999

          ┌─ 'N' ─┐                    ┌─ 'N' ─┐
 time( ───┼───────┼─── , ─ intime ─ , ─┼───────┼─── )
          ├─ 'L' ─┤                    ├─ 'L' ─┤
          ├─ 'C' ─┤                    ├─ 'C' ─┤
          ├─ 'H' ─┤                    ├─ 'H' ─┤
          ├─ 'M' ─┤                    ├─ 'M' ─┤
          ├─ 'S' ─┤                    ├─ 'S' ─┤
          ├─ 'F' ─┤                    ├─ 'F' ─┤
          └─ 'T' ─┘                    └─ 'T' ─┘
       output format                format of intime

-- time('N','9:46am',C)              ⇒    09:46:00
-- time('N',63692300797000000,'F')   ⇒    09:46:37
```

Returns the time format *intime* converted to the requested format.

Date

```
-- Return current date (example for 2019-05-01, 00:00)

                ┌─ 'N' ─┐       ⇒    1 May 2019
 date( ─────────┼───────┼──────────────────────────── )
                │       └─ , ── , ── , ─ sepa ─┘          default separator
                ├─ 'I' ─┤       ⇒    2019-05-01             '-'
                ├─ 'S' ─┤       ⇒    20190501               ''
                ├─ 'E' ─┤       ⇒    01/05/19               '/'
                ├─ 'O' ─┤       ⇒    19/05/01               '/'
                ├─ 'U' ─┤       ⇒    05/01/19               '/'
                ├─ 'M' ─┤       ⇒    May
                ├─ 'W' ─┤       ⇒    Wednesday
                ├─ 'D' ─┤       ⇒    121                (1...366 days)
                ├─ 'B' ─┤       ⇒    737179             (basedate: days since 1 Jan 0001)
                │       │
                │       │       --   Counters:
                ├─ 'F' ─┤       ⇒    63692265600000000  microseconds since 1 Jan 0001 00:00
                └─ 'T' ─┘       ⇒    1556668800         seconds      since 1 Jan 1970 00:00

-- Example of changing separator string:
--      date('E',,,'.')   ⇒    01.05.19
```

For calculations using time format **Full** (*Micro*seconds) or **Ticks** (seconds) the value of NUMERIC DIGITS has to be at least to 18 or 12. See page 16.

Convert Date Formats

```
-- Converting date format for any date from Monday, 0001-01-01 to Friday, 9999-12-31

                ┌─ 'N' ─┐                 ┌─ 'N' ─┐                 ┌─ {default} ─┐
 date( ─────────┼───────┼─ , ─ indate ─ , ┼───────┼─ , ─────────── , ┼─────────────┼ )
                ├─ 'I' ─┤                 ├─ 'I' ─┤                 └─ insepa ────┘
                ├─ 'S' ─┤                 ├─ 'S' ─┤    └─ sepa ─┘    indate separator
                ├─ 'E' ─┤                 ├─ 'E' ─┤    output       if \= default
                ├─ 'O' ─┤                 ├─ 'O' ─┤                 for indate format
                ├─ 'U' ─┤                 ├─ 'U' ─┤
                ├─ 'M' ─┤                 ├─ 'B' ─┤
                ├─ 'W' ─┤                 ├─ 'F' ─┤
                ├─ 'B' ─┤                 └─ 'T' ─┘
                ├─ 'D' ─┤                indate format
                ├─ 'F' ─┤
                └─ 'T' ─┘
             output format

--  Convertig basedate to calendar date:
--  date('I',737179,'B')            ⇒    2019-05-01
--  date('I',737179,'B','.')        ⇒    2019.05.01    -- changing default separator

--  Obtaining weekday as a digit:  0 = Monday ... 6 = Sunday
--  date('B','2019-05-01','I') // 7  ⇒    2  means Wednesday
```

Returns date format *indate* converted to the requested format. By using argument *sepa* a separator character other than the default may be requested. The *sepa* argument must be exactly 1 character long and not be a letter or figure. A null string is accepted and will suppress the use of separators.
Providing *insepa* is only needed if *indate* does not use the default separator for that format.

For heavy duty work of processing this type of data, object classes **DateTime** and **TimeSpan** provide more than 100 methods.

6 Managing Files and Directories

Windows (and Unix) files, from ticker tape times, inherited their non-structure as a continuous stream of characters.[1] Lines have to be marked by the end-of-line character-pair hexa 0D0A (formerly teletype carriage return and line feed, CRLF) in the data. To find out how many lines a file has, the file system must scan it completely. This has become faster by orders of magnitude in current Windows and hardware.
During **L**ine-oriented processing, ooRexx handles the CRLF invisible to the running program. It strips CRLF from the file lines read and adds them to the lines when writing to a file. If processing is **C**haracter oriented, CR and LF are treated as ordinary data and are visible to the program.

6.1 File Management

Test of Existence, Delete

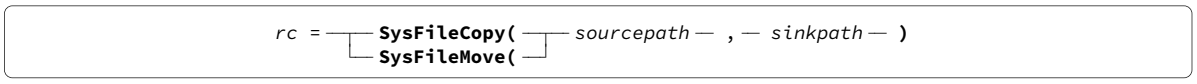
```
flag = SysIsFile( — filepath — )

 0  not found
 1  found

-- Older alternative returns 1 also if filepath is a directory:

flag = SysFileExists( — filepath — )
```

Returns 1 if the file exist, else 0. The older function **SysFileExists()** does the same, but also reports directories. Both do reject place holders like * or ?.

```
rc = SysFileDelete( — filepath — )

 0  deleted
 2  not found
```

Practical for deleting a file, irrespective if it exists or not. Returns the Windows returncode (see page 59), which is 0 if the file was deleted or 2 if not found.

Copy, Move, Rename



Returns 0 after file was successfully copied or moved, else a Windows returncode. **Renaming** a file is done by moving within the same directory.

6.2 Directory Management

If no path is given, files are searched or created in the „current“ directory. This is the directory in which the command to start the ooRexx program was entered. If it is started by double click on an Icon, the entry in the „Execute in:“ field of the Icon Properties dialog applies.

[1] Conversely, mainframes inherited a *record* oriented structure from the Hollerith punch card.

```
qualify( — charstring — )

-- If current directory is:                 'D:\Sandbox'
-- qualify('abstract.txt')          ⇒       'D:\Sandbox\abstract.txt'
-- qualify('.\rexx\abstract.txt')   ⇒       'D:Sandbox\rexx\abstract.txt'
-- qualify('\rexx\abstract.txt')    ⇒       'D:\rexx\abstract.txt'
-- qualify(' ')                     ⇒       ''
```

If *charstring* is a filename, the function returns the full path to the current directory, to which the filename is appended. If *charstring* is an absolute or relative part, it is appended to the drive letter or the current ditectory. If it is a blank or a null string, a null string is returned.

```
flag = SysIsFileDirectory( ┬─ name ──────── )   -- equivalent to .\name
                           ├─ .\name ─────┤     -- subdirectory of current directory
 0  not found              ├─ ..\name ────┤     -- same level as current directory
 1  found                  ├─ ..\..\name ─┤     -- level above current directory
                           └─ x:\path ────┘     -- complete path
```

Returns 1 if the directory exists, else 0.

```
rc = ┬─ SysMkDir( ─┬─ filepath — )
     └─ SysRmDir( ─┘
```

Makes or removes the specified directory and returns 0 if successful or else a Windows returncode.

Reading Elements of Path Names

```
                                                -- Example:  D:\Sandkasten\short.txt
filespec( ┬─ 'D' ─┬ , — filepath — )      ⇒     'D:'
          ├─ 'L' ─┤                      ⇒     'D:\Sandkasten\'
          ├─ 'P' ─┤                      ⇒     '\Sandkasten\'
          ├─ 'N' ─┤                      ⇒     'short.txt'
          └─ 'E' ─┘                      ⇒     'txt'
```

If the requested element does not exist in *filepath*, the null string is returned.

Short Path Names

Some older programs are not prepared to handle blanks in path names. This can be circumvented by using the original „8.3“ short format that Windows creates internally.

```
SysGetShortPathName( — pathlongname — )

-- sysgetshortpathname('C:\Program Files (x86)')   ⇒    'C:\PROGRA~2'
-- sysgetshortpathname('C:\doesnt exist')          ⇒    ''
-- sysgetshortpathname('C:\Users')                 ⇒    'C:\Users'
-- sysgetshortpathname('C:\Benutzer')              ⇒    ''          Explorer GUI quirk

-- reverse direction:
SysGetLongPathName( — pathshortname — )
```

Note that the Explorer GUI in non-English Windows in part uses „translated“ directory names which are unknown to Windows, like German *Benutzer* for the *Users* directory.

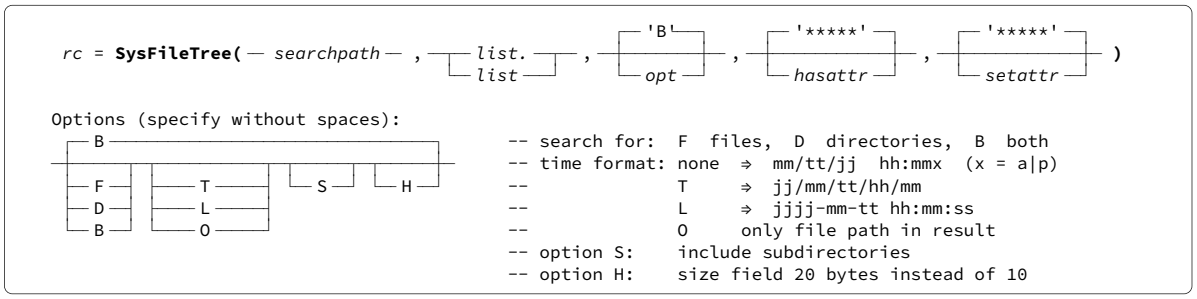
Hint

Windows command **ATTRIB** has the purpose to list or change file attributes. But when searching for a –possibly known only in part– file name, it is a very useful and on current systems fast tool. For example:

```
ATTRIB *1916.pdf /S
```

Searches the current directory and its subdirectories (/S) for all PDF files with a name ending in „1916“. The full pathnames of all hits are written to the screen.
On page 36 it is described how an ooRexx program can process the lines written to the screen. Since *Solid State Disks (SSD)* are replacing mechanical disks, ATTRIB has become markedly faster.

6.3 Searching for Directory- and Filenames



searchpath defines the top directory to be searched and the file names. If no directory is given, the current directory is searched. The usual Windows masks can be used, for example `*.exe`
The search can be limited to files with certain attributes (Archive, Directory, Hidden, Read-Only, System, in short ADHRS). This is defined by using a string *hasattr* of exactly 5 bytes. Three characters are used to indicate the required attribute setting: * for any , + for set, and - for not set (deleted).
The name *list.* defines the destination of the result lines. If it ends –as in this example– in a period, the result will be written to a stem variable[2]. It need not to exist previously. If it does, the existing one will be replaced by the result. Element *list.0* holds the number of the hits found. If this number is larger than 0, elements *list.1 ...* will hold the result in the format shown below.
If the destination name does **not** end in a period (*list*) **and** an array named *list* exists, the result will be written to the array. The elements *list[1]...* will hold the result. If this array does **not** exist, the output will be written to stem variable (*list.*), which is the behaviour of ooRexx versions earlier than 5.0, when output could only be written to a stem variable.
The option characters have to be supplied without spaces, for example 'FLS'.

```
Effect of the time option (none|T|L|O) on the result layout:

....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
none               |19         |31    |38
10/05/19  11:17p       509440  A----  C:\program files (x86)\oorexx\ooDialog.exe

T                 |17          |29    |36
19/10/05/23:17:48       509440  A----  C:\program files (x86)\oorexx\ooDialog.exe

L                      |22          |34    |41
2019-10-05 23:17:48       509440  A----  C:\program files (x86)\oorexx\ooDialog.exe
2015-05-09 16:25:06            0  -D---  C:\program files (x86)\THE\doc     -- Directory!
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8

O
C:\program files (x86)\oorexx\ooDialog.exe
```

If option H is used, the pathnames are moved to the right by 10 positions due to the increased length of the size field.
A 5 bytes long string *setattr* allows to change attributes by providing one of the following characters: * to leave unchanged, + to set, and - to reset. The result returned by **sysfiletree** already shows the changed attributes.

[2] Stem variables are described on page 35; Arrays on page 40.

7 Read and Write Files (New)

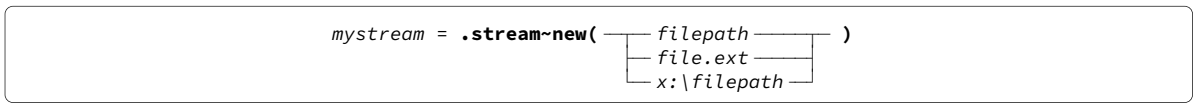
Reading the 3 million lines of a 120 megabytes file on an ordinary Lenovo E595 notebook with mechanical disk using the **~arrayin** method just takes 7.4 seconds. Using the code proposed by *Jon Wolfers* (see below) its only 1.1 seconds.
In our time of large computer memory, files may conveniently read in one single operation into RAM, to be processed in place. This is much faster than the conventional line by line reading from and writing to disk.

- Object type **array** is used for holding the data. How to work with arrays is described starting on page 40.
- An additional requirements is the explicit definition of a **stream** object, acting as a data channel connecting the program and the file.

The conventional way of working with files is described starting on page 30.

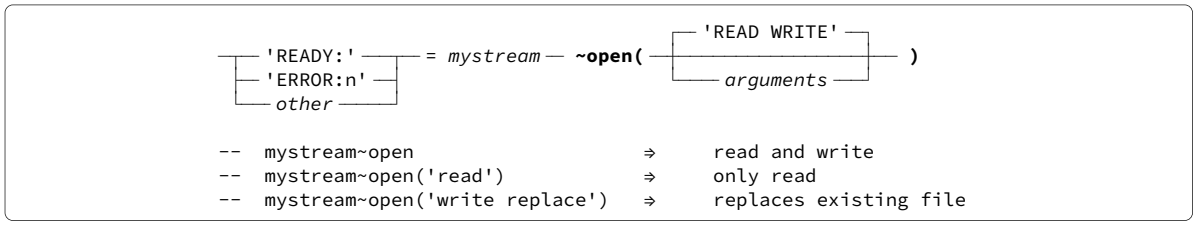
7.1 Copy Data from File to Array

Step 1: Define Stream to Access File



Defines a stream *mystream* through which the file is accessed. The existence of the file or the storage medium is **not** tested at this point.

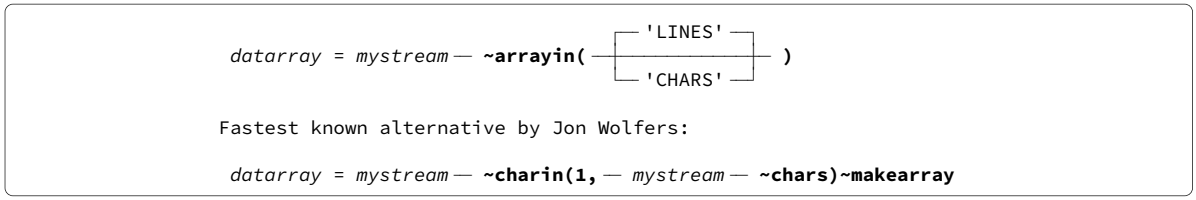
Step 2: Open File



If string `READY:` is returned, the file can be used. In case of an error, a string like for example `ERROR:2` is returned. The code –2 in this example– is a Windows returncode, as described on page 59. Other returned strings like NOTREADY or UNKNOWN are documented but could not be produced during testing for this document.

Step 3: Copy File Data to Array

In a single instruction, the entire file contents is copied to an array object. This places the data in RAM, where it can be processed very fast.



If array *datarray* does not exist, it is automatically created by method **~arrayin**. Should the array already exist, the old data is deleted. In LINES mode, each array item holds one line of the file, the CRLF already being stripped from it.
The shown alternative proposed by Jon Wolfers needs only 1.1 seconds for a test file, while the standard way requires 7.4 seconds.

Step 4: Close Stream

```
mystream — ~close
```

It is good programming style to close a no longer needed file. Otherwise it remains open and the stream blocks resources, particularly if many files are being processed.

If Needed: Informations About a File; Positioning

```
num = mystream — ~lines          ⇒   count of lines still to be read

flag = mystream — ~lines('N')    ⇒   1: lines to be read exist, else 0

num = mystream — ~chars          ⇒   count of bytes still to be read
```

These method calls are self explaining.

```
newpos = mystream ──┬── ~seek( ─────┬── 'string' — )
                    └── ~position( ─┘

--  'string'  has 1 to 3 elements:     -- include string in single or double quotes

                                       -- =1 is start of file;  <0 is end of file (EOF)
                      ┌─ = ─┐          -- = from beginning; =0 is treated as =1
   1.               ──┼─────┼── num ──
                      ├─ < ─┤          -- < from EOF backwards
                      ├─ + ─┤          -- + from current position forward (towards EOF)
                      └─ - ─┘          -- - from current position backwards

   2. (optional)    ──┬──────────┬──   -- use only when different positions for
                      ├─ READ ───┤     -- writing and reading are necesary
                      └─ WRITE ──┘

                      ┌─ C ─┐          -- CHAR: num bytes   (maybe unexpected default!)
   3. (optional)    ──┼─────┼──
                      └─ L ─┘          -- LINE: num lines

--                                        resulting position:
-- mystream~seek('64')                 ⇒  byte 64
-- mystream~seek('=8 L')               ⇒  line  8
-- mystream~seek('< 0 L')              ⇒  beginning of last line
```

If reading is not to start from position 1, method **~seek** provides the required starting position.
The chapter on arrays starting on page 40 describes in detail how to work with the data in an array.

7.2 Copy Data from Array to File

In mode LINES, each array element will be written as a line (with CRLF appended to it) to the file. Because the sequence of four steps is the same as with reading, combining all steps into one display box will suffice:

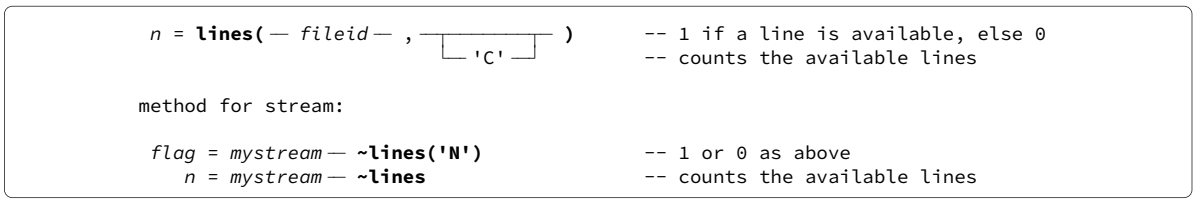
```
  mystream = .stream~new( — outfileid — )    -- 1. Define stream to acces file

  mystream — ~open('write replace')          -- 2. Open file for writing and replacing

  mystream — ~arrayout( — datarray — )       -- 3. Copy data to array (creating new or replacing)

  mystream — ~close                          -- 4. Cleanup after use


-- Faster alternative to step 3:

mystream~charout(datarray~tostring,1)        -- 3a. Proposed by Jon Wolfers
mystream~charout('0D0A'x)                    -- 3b. add missing CRLF of last line
```

In step 3 all array elements are copied in a single operation from RAM to disk. As an **alternative** the steps 3a and 3b may replace step 3.
On closing the file in step 4, method **~close** returns string `READY:`. Should an error occur, a Windows return code is returned, see page 59.
To save a 77 megabytes file of 3 million lines, 9.4 seconds were needed by method **~arrayout**, while step 3a, as proposed by *Jon Wolfers*, needs 2.4 seconds. The file created by step 3a is 2 bytes smaller, because the last line is missing the CRLF pair. This is resolved by step 3b.

8 Read and Write Files (Conventional)

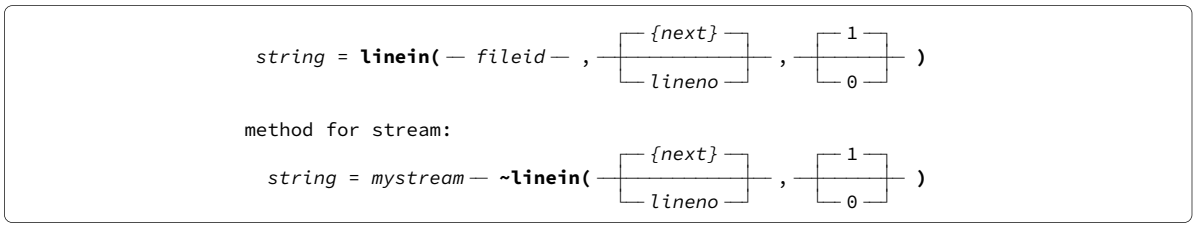
Countless existing Rexx programs make use of the conventional functions. They are simpler to code, because no stream needs to be created. And no explicit open call is required, because it happens automatically on the first access. On the other hand, the conventional functions are noticeably slower when processing large files.
Each call reads/writes exactly one line or one „block“ of data. One block consists of one or more bytes.[1]
Files which are accessed via the new stream approach can also be processed in conventional line and character modes. The method calls for this are also described below.

Get Line Count



Returns 1, if data can be read from *fileid*, else 0. Using argument **C** returns the number of available lines.

Reading File Lines



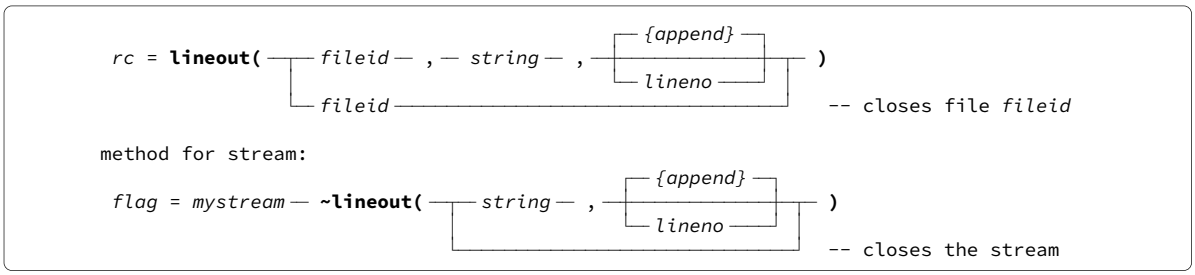
Returns the next line from *fileid*, by default starting with line 1. Only a single line per call is possible. Conceptually, all lines in a disk file are consecutively numbered starting with 1. If argument *lineno* is used, the line with this number is returned. If used, *lineno* may be smaller than the current line number. If a *lineno* is combined with last argument, no data is returned but only the reading position changed. On the next call, the data of line *lineno* is returned.

```
-- Example of a loop for reading a file completely

fileid = 'C:\\demo\\sample.txt'
do i=1 while lines(fileid) = 1   -- now recommended:   do i=1 for lines(fileid,'C')
     ...
     dataline = linein(fileid)
     ...
end i
```

Calling **linein()** after the last line of a file has been read, causes an endless wait. To avoid this, the use of **lines()** is important.

[1] Rexx on the mainframe also uses a command named **EXECIO**, which is available in simplified form. It is described in chapter HOSTEMU of manual *rexxextensions.pdf*.

Writing File Lines

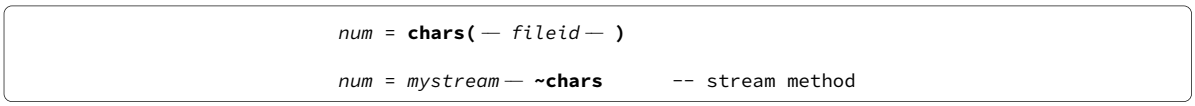


Appends *string* as a new line to file *fileid* and returns 0, in case of error 1. If *lineno* is used, **lineout()** writes the data beginning at the start of line *lineno*. Existing data is overwritten regardless whether old and new length are identical. Line number *lineno* must exist in the file or be the new line at the end of the file.

8.1 Read or Write in Character- or Block Mode

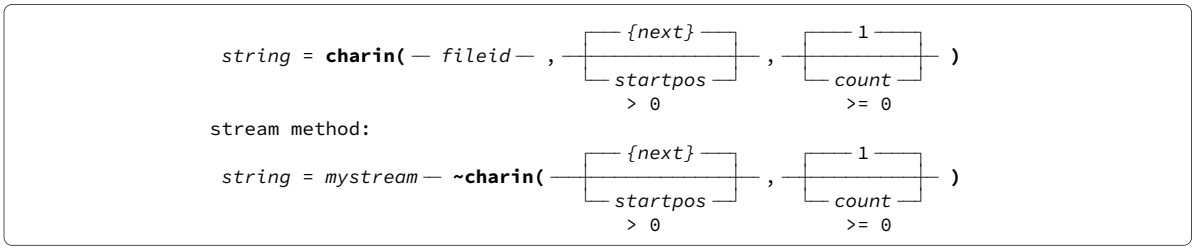
In this mode, the line-end pair CRLF (hexa 0D0A) is not a control sequence but data. It is treated like all other bytes in the file. A block may consist of only 1 byte. A logical line structure is possible, if all lines are blocks of exactly the same length. If, for example, a table has lines (rows) of 244 bytes, all writes or reads must be in portions of 244 bytes or a whole multiple of it.

Get Byte Count



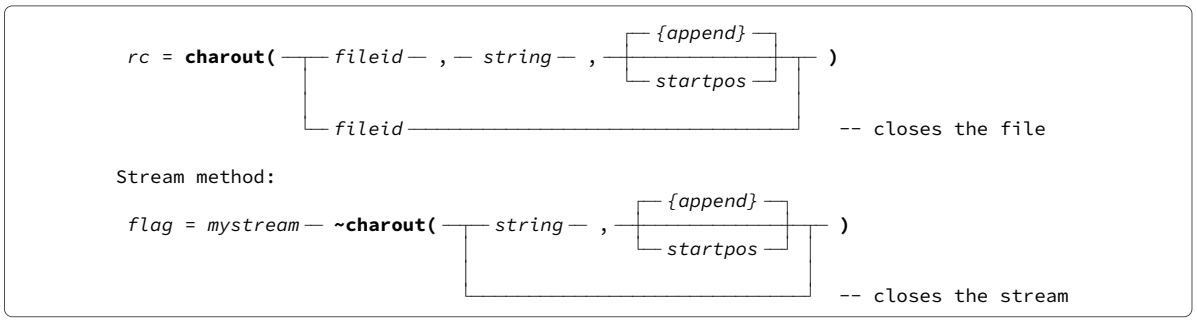
Returns the number of available –not yet read– characters in *fileid* or 0.

Read Bytes



Returns a string of *count* characters from *fileid*. If fewer bytes are available, the string is shorter. By using *startpos*, the reading position may be changed. If combined with last argument 0, only the reading position is changed and a null string returned.

Write Bytes



Appends *string* to the end of *fileid* or overwrites existing data beginning at *startpos*.

Close File

At the end of the program, all files it opened are automatically closed. Good programming practice is closing each file as soon as it is no longer needed.

```
string = stream( fileid , 'C' , 'CLOSE' )
```

Returns the string `READY:` or the null string if *fileid* was not open. In case of error, *string* is the Windows returncode; see page 59. Files which are open for writing can also be closed via **lineout()** and **charout()** as mentioned above.

9 Bits and Bytes

The smallest storage unit is the byte, holding one character and made up by 8 bits. Eight bits of 0 or 1 can have 256 different combinations. Three ways to display a byte are supported by ooRexx, as is illustrated using the letter **Z**:

- The **character** (c) type of display uses the graphical representation (glyph) which we see on the keyboard or on the screen: **Z**. Only a part of the 256 possible values are assigned glyphs that can be directly entered on the keyboard or displayed on the screen.
- In **binary** (b) representation, the 8 bits of character Z are written in ooRexx as `'01011010'b`.[1] The 256 bytes in binary notation span the values:

  `'00000000'b ... '11111111'b`

- The **hexadecimal** (x) representation of character Z is `'5A'x`. It is much easier to understand by a human than a string of bits. The byte is made up of two groups of 4 bits each. Each group has 16 possible values. The first 10 are represented by digits 0 through 9. Higher values are represented by using A through F acting as „figure". A represents 10 and so on. The 256 bytes in hexadecimal notation span the values:

  `'00'x ... 'FF'x`

A fourth type of representation, which unlike the above cannot be directly written as an ooRexx expression, is **decimal** (d) numbers. It is simply the consecutive numbering of the 256 characters from 0 to 255. Letter Z has the decimal value 90. To use this representation, which for example is necessary when handling storage addresses, a conversion step is required.
Binary `'11010110'b`, hexadecimal `'D6'x` and decimal (`214`) are representations of the same Byte. For the character representation this is only partly true. Letters and symbols which are not part of the English language were over time and in different countries assigned to different bytes. Industry created „codepages" in an attempt to define standard assignemts of glyphs (like ö) to a byte value. From country to country, different codepages are in use.[2] Outside of byte range `'20'x ... '7E'x` (blank, digits, letters, punctuation) one has to carefully check what glyph is displayed if different computers or program versions come into play.
When coding binary or hexadecimal strings, ooRexx ignores blanks, which may be used for improved readability. If necessary, ooRexx pads a string with zeros to the length of the next full byte. The following table illustrates the available conversion functions (read „2" as „to"):

from to:	Bits	Char	Decimal	heX
Bits		—	—	b2x
Char (glyph)	—		c2d	c2x
Decimal	—	d2c		d2x
heXadecimal	x2b	x2c	x2d	

Conversion from and to binary is only available via the hexadecimal representation. Binary and hexadecimal functions expect the quoted argument strings **without** an appended `b` or `x`.

[1] Experience has shown that `b` and `x` should not be used as variable names. When one of these is appended to a quoted string, the interpreter wrongly assumes a binary or hexadecimal string.
[2] In German Windows 10, the console uses codepages 850, while Notepad uses 1252. The Hessling Editor (THE) version 4.0 changed its default from 850 to 1252. Hex D6 is shown as glyph Í in 850 and as Ö in 1252.

Conversion Examples

```
c2x('Z')           ⇒    5A
x2c('5A')          ⇒    Z
x2b('5A')          ⇒    01011010
b2x('0101 1010')   ⇒    5A
```

Positive Whole Numbers

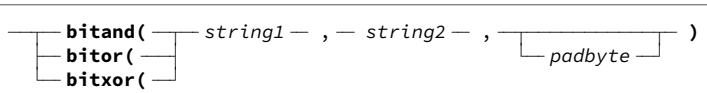
```
c2d('Z')      ⇒    90
x2d('FFFF')   ⇒    65535
d2x(123456)   ⇒    1E240     -- note not 01E240 (no leading 0)
```

Negative Whole Numbers

```
x2d('FFFF',4)   ⇒    -1         --  4 halfbytes = FFFF
x2d('FFFF',8)   ⇒    65535      --  padded to 8 halfbytes = 0000FFFF
x2d('00FF',2)   ⇒    -1         --  length from the right hand end: hex FF
x2d('FFFF',0)   ⇒    0
d2x(-1,2)       ⇒    FF
d2x(-1,8)       ⇒    FFFFFFFF
```

Negative numbers always require a length argument. The first bit is a sign bit.

Bitwise Logical Operations




```
Examples:
      e  =  '65'x   =   '0110 0101'b
      Y  =  '59'x   =   '0101 1001'b
                        ─────────────
  bitand('e','Y')    ⇒   '0100 0001'b  =  '41'x  =  A
  bitor('e','Y')     ⇒   '0111 1101'b  =  '7D'x  =  }
  bitxor('e','Y')    ⇒   '0011 1100'b  =  '3C'x  =  <
```

These functions return the string that results from applying the logical AND, OR or XOR bit by bit to *string1* and *string2*. If *padbyte* is supplied, the shorter bitstring is padded with it. Otherwise, the logical comparison stops at that point and the remaining bits of the longer string are copied to the output.

10 Multitool: Stem Variable

The most simple data collection, called stem variable, already existed in REXX on the mainframe. Countless existing programs make use of it. Its multidimensional variant is often called *compound variable*.
The *name* of a stem variable has at least two parts (stem and index), divided by a period:

```
stem.branch
stem.branch.twig
stem.branch.twig.leaf
and so on ...
```

A short list of some towns that were members of the medieval *Hanseatic League* will serve as an example. The stem of this variable is `hanse.` and whole numbers are used as index:

```
hanse.1  = 'Bremen'
hanse.2  = 'Hamburg'
hanse.3  = 'Lübeck'
hanse.4  = 'Wismar'
hanse.5  = 'Rostock'
hanse.6  = 'Stralsund'
hanse.7  = 'Greifswald'
```

This kind of stem variable with numerical index is by far the most used, because it serves as a vehicle to exchange data between an ooRexx program and subprograms it calls. Its importance and usefulness in this role has increased a lot, when keyword **use arg** was introduced in Rexx 3.0 (see page 47). This made it possible for two programs to *share* a big data collection **without** large internal copying operations.

```
i = 4
say hanse.[i-1]     ⇒     Lübeck
say hanse.i         ⇒     Wismar
say hanse.[i+3]     ⇒     Greifswald
```

Enclosing the index expression in brackets [] simplifies accessing neighbouring elements, as is often required in loops. It also makes possible to have the index value in another stem variable, which otherwise would not work.

```
index.999  =  5
say hanse.[index.999]    ⇒     Rostock
```

10.1 Enumerated Stem Variable

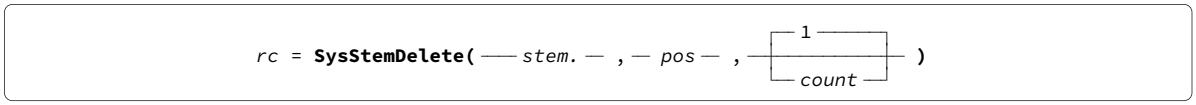
This is the conventional way for a called program to return its data to the calling program. *Enumerated* adds the following rules to the numerical index:

- The first element has number 1.
- The following elements are *consecutively* numbered by adding 1.
- Element 0 contains the number of the last existing element. In our Hanseatic League example it is: `hanse.0 = 7`.

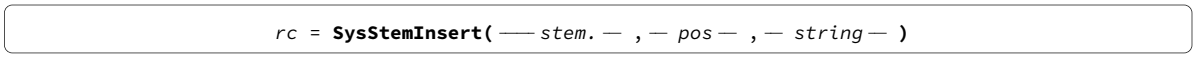
In this Short Reference, a stem variable that has these properties is called *enumerated stem variable*.
Internally, ooRexx does **not** depend on the above properties, as we will see later. The *enumerated stem variable* is purely a convention obeyed by the classic interface. Examples are functions like **SysFileTree** (see page 26) and the four functions described below.

Utility Functions for Enumerated Stem Variables

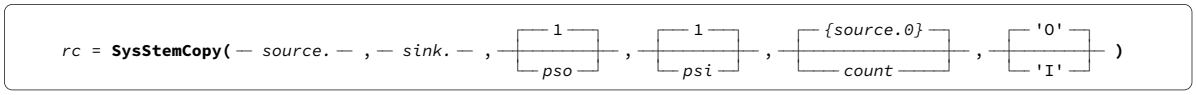
In all these function calls, the *stem.* argument may be written without the period. Element *stem.0* is set by these functions as required.



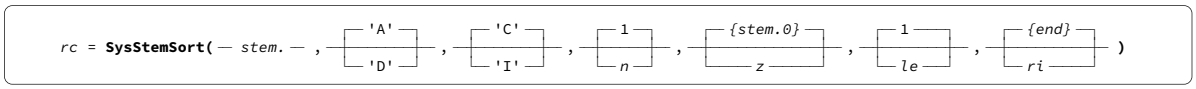
Deletes exactly *count* elements from *stem.* starting with element *pos*. The elements coming after the deleted ones fill the gap. If element 3 is deleted, former element 4 will be assigned index 3 and counter *stem.0* is reduced by 1.



Adds *string* as a new element with index *pos*. An existing element at *pos* and all following get their index increased by 1. To append *string* as a new element, *pos* must be set to *stem.0* plus 1.



Starting at position *pso*, copies exactly *count* elements from *source.* to *sink.*. Default for *count* is all elements of *source.* Existing elements in *sink.*, starting at position *psi* are overwritten.[1] If the sixth argument is **I**, the copied elements are inserted starting at position *psi* and the old elements are assigned higher index numbers accordingly. To append the elements to *sink.*, position *psi* must be set to *sink.0* plus 1.
If stem variable *sink.* does not exist, it will be a complete copy of *source.*, provided no other arguments are used.
Special case, if *sink.* exists and **Insert**-mode is used and no other arguments are used: The elements copied from *source.* will precede(!) the old elements of *sink.*.

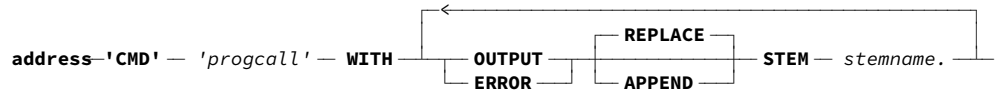


Sorts the elements in *stem.* based on the characters which each element contains at the columns *le* trough *ri*. Default for *ri* is the end of the element. By default, sort is in ascending (A) order, observing upper- and lowercase (C) of letters. Argument D creates a descending sort order. Argument I ignores the case of letters. The sorting may be limited to elements *n* through *z*. Default is sorting all elements. The sorting algorithm used is „not stable“. Elements with *identical* will **not** keep the original sequence they had before the sort. See page 43 for an alternative.

Reroute Screen Output to a Stem Variable

Sometimes it would be very useful to be able to somehow „catch“ the screen output of external programs –like `pdftk.exe` in the following example– to be able to process it. Beyond Windows Pipelining not much can be done, because compiled programs cannot be changed.
Since ooRexx 5.0, screen output that an external program produces using STDOUT and STDERR can be re-routed to an enumerated stem variable. The calling ooRexx program uses extension **WITH** of keyword **ADDRESS** for this. The screen output is then available to it as an enumerated stem variable. This works for EXE-, BAT- and CMD-programs as well as existing Rexx programs.

[1] Contrary to what the Reference says, the default is **O**verlay, not **I**nsert according to my test.




```
-- Example: Reroute of screen output from PDFTK.EXE to stem variable HELPINFO.
 address 'CMD' 'pdftk.exe --help' WITH output stem helpinfo.   -- sets helpinfo.0 = 428
```

CMD is in quotes here because of the **ADDRESS** syntax. It passes *progcall* on to Windows, which starts the external program in question. It is strongly recommended to put *progcall* also in quotes to avoid interference between external command syntax and the ooRexx interpreter. If it contains blanks, *progcall* must be in quotes.
Keyword **WITH** is followed by one or two instruction blocks. If used, keyword **OUTPUT** with **STEM** controls re-routing STDOUT to stem variable *stemname.*. Keyword **ERROR** with **STEM**, if used, does the same for STDERR. If both are used, each must be given **its own** stem variable name.
After control returns from the called program, element 0 of each stem variable holds the number of re-routed lines. A previously existing stem variable is replaced.
If option **APPEND** is used, element *stemname.0* must exist and contain a number *count*. The new lines are then written starting with element *stemname.[count+1]* and so on.
The extension **WITH** of keyword **ADDRESS** offers more, rather complex processing. For example, data can be supplied to STDIN. I did not see any functionality which I had been missing. Therefore I did not test further and cannot describe the other processing modes.

Handling Blanks in Path Names

Some older programs cannot handle blanks in path names correctly. Windows internally creates for each directory and file a short name in the legacy 8.3 format. See page 25 for a function to get this short name.

10.2 General Stem Variable

As already mentioned, ooRexx itself is not dependent on enumerated stem variables. It neither needs starting with 1 nor are gaps in the indexing a problem. Our example of Hanseatic League towns could also be implemented with German telephone area codes:

```
                                hanse.0421    = 'Bremen'
                                hanse.040     = 'Hamburg'
                                hanse.0451    = 'Lübeck'
                                hanse.03841   = 'Wismar'
                                hanse.0381    = 'Rostock'
                                hanse.03831   = 'Stralsund'
                                hanse.03834   = 'Greifswald'
```

It is important to note that stem indexes are in fact *character strings*, not numbers to ooRexx. Index 040 is *not* the same as 40.

```
                    wanted = 03831
                    say 'Hansestadt' hanse.wanted    ⇒    Hansestadt Stralsund
```

If ooRexx encounters a stem variable –in this case `hanse.wanted` – the index is checked for being a variable having a value. Here, this is actually true: `wanted` is a variable having value `03831`.
Next, ooRexx checks if the resulting stem variable `hanse.03831` is assigned a value. This again is true; the value is `Stralsund`. The interpreter replaces the stem variable in the code line with this value, resuling in the screen output shown above.
If an undefined area code is used, the following happens:

```
                   wanted = 0815
                   say 'Hansestadt' hanse.wanted    ⇒    Hansestadt HANSE.0815
```

The resulting stem variable is `hanse.0815` in this case. It has no assigned value and is treated like any unknown variable (see page 55) as „itself“ folded to uppercase: `HANSE.0815`.

Initialize

To avoid an output like `HANSE.0815` it is possible **before** the first element of a stem variable is assigned, to define a default value. This will be used when a non-existing index of the stem is referenced. For stem `hanse.` this could for example be:

```
hanse.   = 'unknown area code'
```

As a result, *unknown area code* is used as value of undefined `hanse.0815`. The null string may also be used. Warning: If a non-empty stem is initialized, all existing data in the stem is deleted.

Character String Indexing

Because the index is a string, it may use not only digits, but in principle all characters that have no special function in ooRexx. Experience has shown that the use of special characters may result in reduced readability and may have unexpected side effects when editing the file. In the following example, we use letter groups from German motor vehicle license plates. The codes for the Hanseatic League towns are:

```
hanse.     = 'unknown'
hanse.HB   = 'Bremen'
hanse.HH   = 'Hamburg'
hanse.HL   = 'Lübeck'
hanse.HWI  = 'Wismar'
hanse.HRO  = 'Rostock'
hanse.HST  = 'Stralsund'
hanse.HGW  = 'Greifswald'
```

Warning: The above assignments are only coded that way to show the principle. See the paragraph after the next to learn the reason.

```
wanted = 'HST'
say 'License from' hanse.wanted    ⇒    License from Stralsund

wanted = 'EMM'
say 'License from' hanse.wanted    ⇒    License from unknown
```

Apart from using letters, the principle is the same as with the area codes.

```
licen = 'HB'
hanse.licen = 'Bremen'
licen = 'HH'
hanse.licen = 'Hamburg'
 etc.
```

By putting the short letter groups in quotes, the intended result is produced. Doing the coding as shown first, will result in unwanted effects if one of the index names HB ... HGW already exists as a variable. In particular, later changes in the program are prone to overlook such things and cause unexpected behaviour.
In my experience, this safe coding by using quoted strings is not an obstacle in practice, because the relevant data is mostly read from files, not coded manually.

10.3 Multidimensional Stem Variable

```
-- NL for Nederlanden (Provincien)
country = 'NL'
prov    = 'GE'
land.country.prov = 'Gelderland'
prov    = 'GR'
land.country.prov = 'Groningen'

-- DE for Deutschland (Bundesländer)
country = 'DE'
bland   = 'MV'
land.country.bland = 'Mecklenburg–Vorpommern'
bland   = 'NI'
land.country.bland = 'Niedersachsen'
bland   = 'SN'
land.country.bland = 'Sachsen'

-- FR for France (Departements)
country = 'FR'
dept    =  72
land.country.dept = 'Sarthe'
dept    =  62
land.country.dept = 'Pas-de-Calais'

-- Application example:
ix = 'FR'
iy =  72
say country.ix.iy      ⇒   Sarthe
```

This example shows the principle of how to use a stem variable with two dimensions (also called *compound variable*). The first index (branch of the stem) is a country code. The second index (twig of the branch) is a code for the administrative subdivisions of each country.
The example also shows that numerical and general indexes can be used at the same time. A numerical index need not be quoted, because anything starting with a digit cannot be a variable.
For multidimensional stem (compound) variables, default initializing by using an assignment of the type `land. = xyz` is **not** possible.

11 Multitool: Array

Since ooRexx 5.0 there are 13 data collection classes defined. Of these, **.array** stands out because of its role in significantly faster reading and writing of files. This was the main reason for me to include it in this *Classic* Short Reference. Another quite useful feature are the methods for direct handling of used and unused index locations.
Also, the performance is improved. I converted a program processing 3 million lines, each holding an X,Y coordinate pair, from using stems to arrays. This reduced run time (excluding disk access) from 55.5 to 38.5 seconds or by 30 percent.

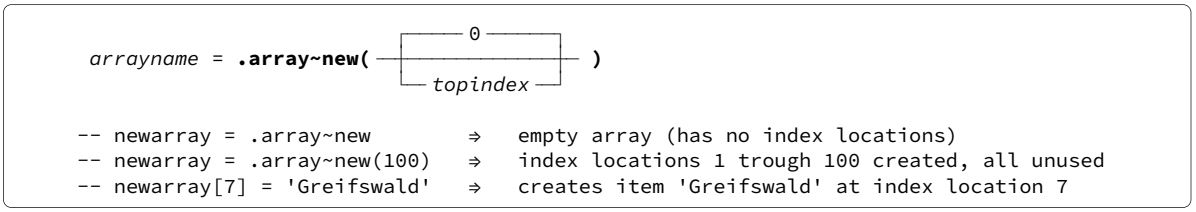
Common and Different Properties of Stem Variable and Array

Common to both collections is the structure of an **index** to access **items**, which contain data. A stem variable uses **character strings** as index. Although practical use of enumerated stem variables is dominant, the indexes are really strings. 40 and 040 define *different* index locations. At the same time, using strings as index (license plate codes, for example) offers possibilities that arrays cannot offer.
An *array* only uses the **positive whole numbers** 1, 2, ... as index. 40 and 040 do refer to *the same* index location. Index location 0 does not exist. An array keeps internal counters for used and unused locations, which are accessible through method calls.
If a multi-dimensional array is created by defining the maximum size of each dimension, all its elements can be initialized by a single method call. This is something a multi-dimensional stem variable does not allow.

```
-- Stem variable:
 mystem.6   = 'Stralsund'          -- usual notation
 mystem.[7] = 'Greifswald'         -- alternative notation
 say mystem.[i+2]                  ⇒ Greifswald   -- assuming: i = 5
 mystem.['HRO'] = 'Rostock'        -- index is not limited to digits
 mystem.4.6.2                      -- using a 3-dimensional stem variable
 parse var mystem.i ...            -- use parse syntax for variables
 mystem~isA(.stem)                 ⇒ 1 (else 0)   -- name here without period

-- Array:
 hanse = .array~new                -- array must be defined before use
 hanse[7] = 'Greifswald'           -- notation is without period
 say hanse[i+2]                    ⇒ Greifswald   -- assuming: i = 5
 say hanse[0007]                   ⇒ Greifswald   -- 0007 = 7 numerically
 hanse[4,6,2]                      -- using a 3-dimensional array
 parse value hanse[i] with  ...    -- different parse syntax required
 hanse~isA(.array)                 ⇒ 1 (else 0)
```

Overview diagram of –in part subtle– notation differences. Method **~isA** tests the collection type.
To keep the description easy to understand, emphasis in this chapter is on a one-dimensional array of character strings.

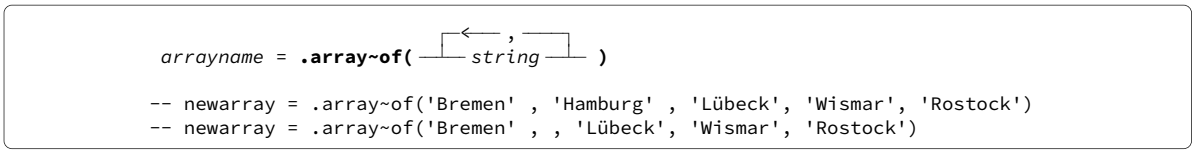
Create an Array and Fill with Data



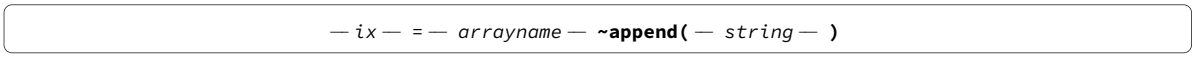

```
-- newarray = .array~new        ⇒   empty array (has no index locations)
-- newarray = .array~new(100)   ⇒   index locations 1 trough 100 created, all unused
-- newarray[7] = 'Greifswald'   ⇒   creates item 'Greifswald' at index location 7
```

This method defines array *arrayname* with zero index locations. Which is called an empty array.

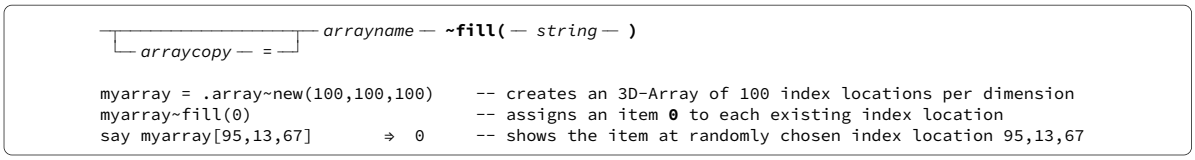
In the second example, the defined single dimension array has 100 index locations predefined, but without associated items yet. Index locations exist, but are „unused“. The created location count is not fixed. Index locations beyond 100 may be added at any time. On the other hand, the number of dimensions cannot be changed.
The third statement assigns to index location 7 an item containing the character string „Greifswald“. Index location 7 now is „used“.
When an item for index location 7 is created in an otherwise empty array, the lower index locations 1 through 6 are automatically created. Array size is 7. Index locations 1 through 6 are „unused“.



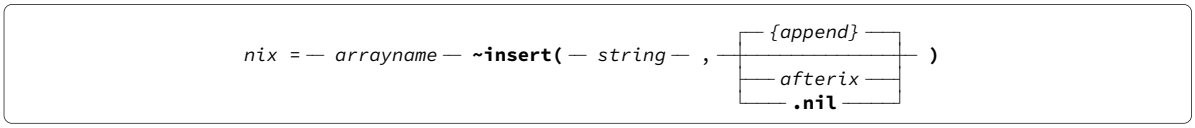
Creates an array and immediately uses the index locations for items. In the first example, 5 arguments exist, which results in index positions 1 through 5 being created and used, holding the strings of arguments 1 through 5.
In the second example, argument 2 is empty (nothing between the commas). Consequently, index location 2 remains unused.



This method operates independent of the size of the array. The location of the last **used** index is determined and *string* becomes the item of the next location. The new index location is returned; which will be 1 if the array was empty or all index locations were unused.



All existing index locations in *arrayname* –no matter if they are used or not– receive *string* as new item value. A copy of the resulting array can be obtained as return value.

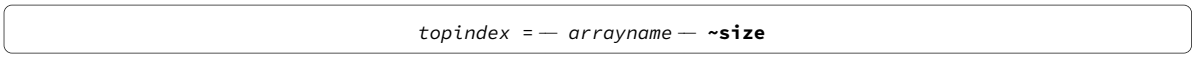


Inserts *string* as a new item after index location *afterix*. The resulting index location *nix* is returned. To insert in front of index location 1, enter **.nil** (see next section) in place of *afterix*. If no insert location is given, this method works exactly like **~append**.

Filling an array with data from a file is done by method **~arrayin** of the **.stream** class. This was already described on page 27.

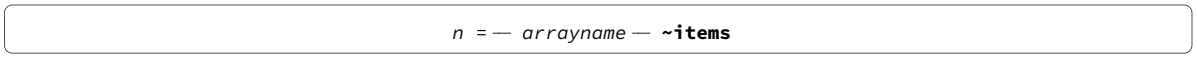
Information about an Array

The legacy way to report condition „No data exists“ in ooRexx is the *null string*. Since ooRexx 3.0, the special object **.nil** may report this condition. In the program code, its name `.nil` may written directly. An example was shown in the description of method **~insert** above. It is possible to assign value `.nil` to a variable.
In places where ooRexx normally writes character strings (screen output, writing files etc.), this object is replaced by the text „The NIL object“.

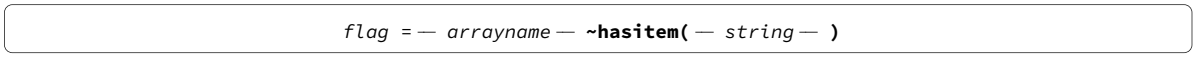


Returns the highest existing index location –no matter if used or not– or 0 if empty.

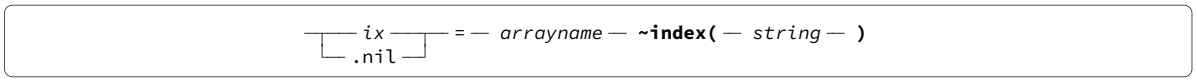
If *dim* is not specified, the number of dimensions is returned, which is 0 for an empty array. If dimension *dim* is specified, its size is returned. If that dimension does not exist, 0 is returned.



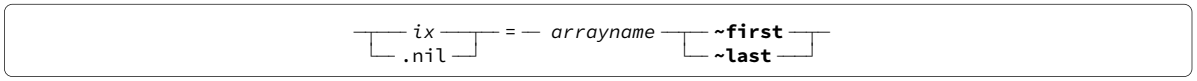
Returns the number of *used* index locations (those with an item), which may be 0.



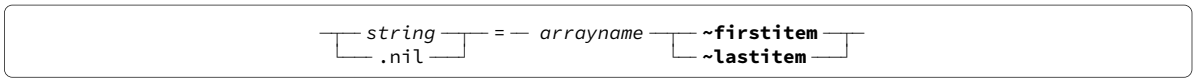
Using strict comparison **==** (see page 56), searches for *string* being an item anywhere in the array. If YES, 1 is returned, else 0.



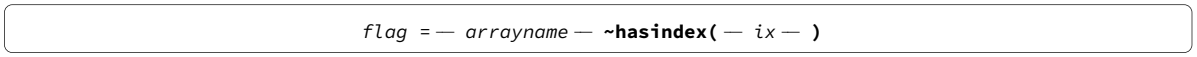
Does the same search as method **~hasitem**, but returns the **index location**, if found. Else **.nil** is returned. In case of multiple occurrences, the first index location is returned.



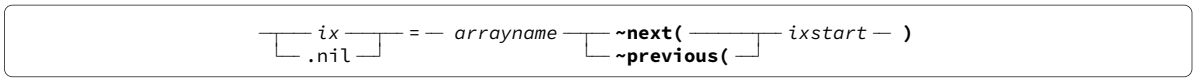
These methods return the lowest/highest *used index location*. If all are unused, **.nil** is returned.



These methods return the *string* that is the item of the lowest/highest index location used. If none exists, **.nil** is returned.

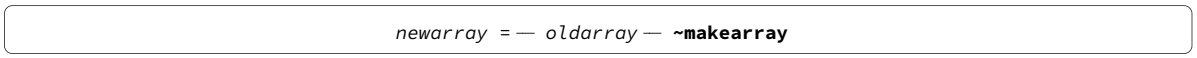


Returns 1 if index location *ix* is used, else 0.

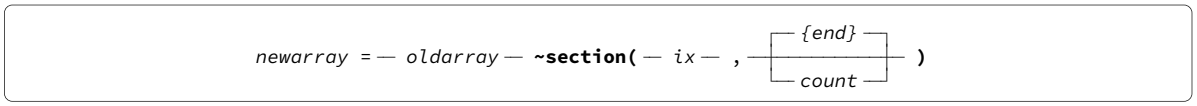


Starting from index location *ixstart*, the next/previous *used* location is returned. If in the requested direction no item exists, **.nil** is returned. The starting location *ix* may be beyond the array size. In this case **~next** always returns **.nil**, while search **~previous** starts at the array size.

Copying an Array (Special Cases)



Returns a one-dimensional array which contains all items from *oldarray*, keeping the same order. Unused index locations are not copied.



This method requires *oldarray* to be one-dimensional. It copies index locations,[1] including the items of used locations. The copy starts at index location *ix* of *oldarray*, which becomes index location 1 of *newarray*. The number of copied index locations is *count*, which becomes the size of *newarray*. The default is all index locations to the end, which also applies if *count* exceeds this number.

[1] The Reference text (p. 258) seems to limit the operation to „items“, but that was not the outcome of my test.

Deleting Data from an Array

```
'' = — arrayname — ~empty
```

Deletes all items from the array, changing all index locations to unused status. Returns a null string.

```
┌── string ──┐
└── .nil ────┘ = — arrayname — ~delete( — ix — )
```

Index location *ix* is deleted. If it is in use, the associated item is returned, otherwise **.nil** is returned. All locations in the array following the deleted location are moved 1 up. If the index location does not exist, nothing changes and **.nil** is returned.

```
                                  ┌─<─ , ─┐
string = — arrayname — ~remove( ──┴── ix ──┴── )
```

Returns the item at index location *ix* and removes it from the array. The index location becomes unused, but is not deleted. If the location is unused, **.nil** is returned.[2]

```
string = — arrayname — ~removeitem( — string — )
```

Works the same as **~remove**, but searches for an item equal to *string* to remove it. Strict comparison **==** is used. In case of multiple occurrences, the first item found is removed.

11.1 Stable Sorting of Arrays with SORT2

Elements that *compare equal* are left untouched by a *stable* sort algorithm. They keep their exact sequence from before the sort. This may for example be important when analysing log data.
The external library **rgf_util2.rex**[3] in its function **sort2** offers sorting capabilities far beyond the capabilities of the function **SysStemSort** (page 36) or the method **SortWith**:

- correct sorting of numbers according to the numeric value, independent of notation,
- sorting by **multiple** fields within the same item, and
- at the same time combining ascending and descending sequence.

```
before     <---  sorted ascending  --->
sort        by character    numerical
 10.1          4.2            -7.2
 -0.6          9.5            -0.6
 -7.2         +8.8             4.2
 +8.8         -0.6            +8.8
 10.2         -7.2             9.5
  9.5         10.1            10.1
  4.2         10.2            10.2
```

The usual character by character sort does not handle signed numbers correctly. **Sort2** does this and also handles *non-aligned* numbers, as long as they remain within the given field limits. This also applies to exponential notation, where `960.2` and `9.602E2` have the same value.
A possibly existing (legacy) stem variable can in simple steps (see page 45) be copied into an array, the array sorted and afterwards copied back to the stem variable.

[2] Contrary to what the ooRexx Reference on p. 257 says and the diagram shows, in my tests only a single index location was accepted, or return code 93.926 would stop the program.
[3] Published by *Rony G. Flatscher*, professor at Wirtschaftsuniversität Wien, originally in 2009.

Array Before Sorting

```
meteo = .array~new
--          n Name of Station    Alti  Pressure  Temp
--                                 m    mbar     degC
--          ....+....1....+....2....+....3....+....4..
meteo[1] = '1 Aigle               381   972.0    10.1'
meteo[2] = '2 Col du St-Bernard  2472   751.8    -0.6'
meteo[3] = '3 Jungfraujoch       3580   654.9    -7.2'
meteo[4] = '4 Koppigen            485   960.3    +8.8'
meteo[5] = '5 Neuchatel           485   9.602E2  10.2'
meteo[6] = '6 Waedenswil          485     960.1   9.5'
meteo[7] = '7 Zermatt            1638   834.7     4.2'
```

The items in array `meteo` are sorted by station name. This data is from *MeteoSwiss*.

SORT2 Syntax

Sort2 is called as a function (not as a method), but expects an array as input. Apart from sorting this array in place, it also returns a new array as result.

```
newarray = sort2( — datarray — , ┌──<──, ──┐ — sortfield — ┘ )

sortfield:  — start — , ┌ {all} ┐ , ┌ 'A' ┐ , ┌ 'I' ┐
                        └ length ┘   └ 'D' ┘   ├ 'C' ┤
                                               └ 'N' ┘
-- By default, also datarray  is sorted.
-- If datarray is not to be sorted, append method ~copy to its name.
```

Argument *datarray* is the array to be sorted. Each **item** in this array is a line of data (character string).
The next argument is *sortfield*, which consists of at least 1 and up to 4 comma separated sub-arguments. These define the sort columns and the sort type. Multiple *sortfield* arguments are possible, each separated by a comma.
The first sub-argument of *sortfield* is *start*, defining the first column of the sort key. This is followed by sub-argument *length*, defining the last column[4]. Default is to *end of item* (line).
The next sub-argument controls the order: **A** (ascending) or **D** (descending).
The last sub-argument controls the comparison. **I** ignores upper- and lowercase, while **C** (case) respects it, **N** activates numerical comparison.
After processing, source array *datarray* is sorted. If it is desired to keep it unsorted, method **~copy** can be appended to the name: *datarray~copy*. In this case, **sort2** uses a temporary array as source.[5]
Function **Sort2** always returns a new array object *newarray*, which is a copy of the sorted *datarray*. An existing array is replaced.
If syntax **call sort2 datarray , ...** is used (see page 57), system variable RESULT becomes the new array.

Example 1: Sorting by Character

The problem is to sort the items of array `meteo` in **descending** order of station *altitude*. This data is located in columns 22 to 25 of each item. The sort key therefore starts in column 22 and is 4 columns long. Sub-argument D creates descending order.
Because the altitude numbers are right aligned and have no sign, it is not necessary to activate numerical comparison; sorting by character will do. Because the last sub-argument is not used, default mode I is active. Function call and result are shown below:

[4] This is in common with ooRexx functions, but different from sort key specification in **SysStemSort** and the Hessling-Editor, which use the last column.
[5] This creates an additional array. When sorting large arrays, it should only be used if program logic requires to keep the unsorted version of *datarray* active. Keyword **DROP** can be used to delete an array and free its storage.

```
 sortmeteo = sort2(meteo~copy,22,4,'D')

--  created sorted array sortmeteo
--  3 Jungfraujoch       3580   654.9    -7.2
--  2 Col du St-Bernard  2472   751.8    -0.6
--  7 Zermatt            1638   834.7     4.2
--  4 Koppigen            485   960.3    +8.8
--  5 Neuchatel           485   9.602E2  10.2
--  6 Waedenswil          485     960.1   9.5
--  1 Aigle               381   972.0    10.1
```

The sort result has been stored in new array **sortmeteo**. As mentioned above, source array **meteo** would also have been sorted. This has been prevented by appending **~copy** to its name.
Note that the 3 stations at 485 m have kept their relative order (stable).

Example 2: Numerically Correct Sorting

```
drusort = sort2(meteo,22,4,'D' ,,28,9,'A','N' )

--  sorted array drusort
--  3 Jungfraujoch       3580   654.9    -7.2
--  2 Col du St-Bernard  2472   751.8    -0.6
--  7 Zermatt            1638   834.7     4.2
--  6 Waedenswil          485     960.1   9.5
--  5 Neuchatel           485   9.602E2  10.2
--  4 Koppigen            485   960.3    +8.8
--  1 Aigle               381   972.0    10.1
```

In this example, an additional second sort key will sort stations of the same altitude by barometric pressure ascending. Numerical comparison is required, because the pressure figures are not all aligned by decimal point and exponential notation is also encountered. The sort key spans columns 28 to 36, or 9 columns long. The returned array is **drusort**.
Two commas are preceding start column 28. Because further separator commas now follow, all 3 commas of *sortfield* argument syntax must be present (first comma). The second comma separates the first *sortfield* (by altitude) from the next (by pressure).
The result shows the stations with identical altitude (485 m) in the desired ascending order of air pressures, as if written 960.1, 960.2 and 960.3.

From Stem Variable to Sorted Array Object and Back

In already existing Rexx programs, possibly a lot of code may exist, written to process enumerated stem variables (page 35). If changes must be kept to a minimum, a possible solution is to make *intermediate* use of an array, just to use the new sort capabilities.

```
/*         n Name of Station    Alti  Pressure  degC  */
/*         ....+....1....+....2....+....3....+....4.. */
zeile.1 = '1 Aigle               381   972.0    10.1'
zeile.2 = '2 Col du St-Bernard  2472   751.8    -0.6'
zeile.3 = '3 Jungfraujoch       3580   654.9    -7.2'
zeile.4 = '4 Koppigen            485   960.3    +8.8'
zeile.5 = '5 Neuchatel           485   9.602E2  10.2'
zeile.6 = '6 Waedenswil          485     960.1   9.5'
zeile.7 = '7 Zermatt            1638   834.7     4.2'
zeile.0 = 7
```

We use the same data from *MeteoSwiss*, this time coded as stem variable **zeile**.

```
meteo  = .array~new          -- create array meteo

do i=1 to zeile.0            -- index 1...7
     meteo[i] = zeile.i      -- copy data to elements
     end i
```

The loop copies the stem variable into array **meteo**, which may immediately be sorted with the capabilities of **sort2** described in the previous section.
Once we have the result in array **drusort**, there are two basic ways to copy the data back into the stem variable for further processing.

```
-- Copy sorted array drusort  back to
-- the enumerated stemvariable zeile.

-- alternative 1: conventional loop

do i=1 to zeile.0
     zeile.i = drusort[i]
     end i
```

The above solution is a conventional loop with an iteration variable. It requires to know explicitly the number of items in the array. As is the convention for enumerated stem variables, this number was stored in element **zeile.0** at the beginning.

```
-- alternative 2: DO ... OVER loop
-- using COUNTER keyword (i = 1, 2, ...)

do counter i elem over drusort
     zeile.i = elem
     end elem

-- alternative 3: DO WITH ... OVER loop
-- using INDEX and ITEM keywords

do with index i item elem over drusort
     zeile.i = elem
     end
```

Alternatives 2 and 3 use the new loop types intended for processing data collections (see page 14). Two simple variables are needed, their values will be set by ooRexx when it processes item by item in the loop. The first, which is named **elem**, holds the item data of the current iteration. The other, which here is named **i**, holds the index number processed in the current iteration. It is needed for the correct stem index. In alternative 2, **i** is defined via the COUNTER keyword, which increments it on each iteration. This guarantees consecutive numbering of the stem elements, even if there are unused index locations in the copied array.
In alternative 3 the ITEM keyword sets **i** to the currently processed index location. If there are unused index locations in the array, the corresponding stem variables will remain untouched by the loop code.

Downloading Library rgf_util2.rex

The current version of this library is part of BSF4ooRexx (see page 59). The file is in the top directory `\bsf4oorexx` of the installation ZIP file.
Alternatively, the file can directly be downloaded from *Sourceforge* using the first link of the following list. Note that **850** is the BSF4ooRexx version number at the time of writing and will change in the future.
The other three links are documentation provided by Wirtschaftsuniversität Wien.

```
sourceforge.net/p/bsf4oorexx/code/HEAD/tree/branches/850/bsf4oorexx.dev/bin/rgf_util2.rex

wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_RGF_UTIL2-20100120-refcard.pdf
wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_RGF_UTIL2-20100806-article.pdf
wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_rgf_util2.pdf
```

Any ooRexx program that wants to use functions from this library needs the following directive, added at the end(!) of the program file:

```
::REQUIRES rgf_util2.rex
```

12 Multitool: USE ARG

In place of making 1000 calls to a subprogram, it may be possible to achieve the same result faster by making only 1 call. The trick is to give the subprogram direct access to the 1000 data items that need to be processed.
This can be done by using keyword **USE ARG**, which allows **shared** access to large data collections without creating additional copies. This is possible, if the data resides in stem variables or arrays.
The main program uses a stem or array name as a call argument. In the called program, keyword USE ARG can access the data. Not only the data but also its internal management information is available to the called program. A simplified process may illustrate the principle:

- A main program creates an array of accounts.
- It calls a subprogram with the array name as argument.
- The subprogram reads the argument with USE ARG. This gives it access to the array.
- The subprogram goes through the array, computes the interest for each account and adds it to the account.
- When the subprogram ends, control is returned to the main program.
- The main program finds each account updated with the interest.

The subprogram can change data as well as add or delete elements. For arrays, the methods for handling unused index locations are available. For stem variables, stem.0 has no special function and non-numerical indexes may be used.

```
-- Program code in mainprog.rex
testvar  = 'Ostsee'
hanse.1  = 'Bremen'
hanse.2  = 'Hamburg'
hanse.3  = 'Lübeck'
hanse.4  = 'Wismar'
hanse.5  = 'Rostock'
hanse.6  = 'Stralsund'
hanse.7  = 'Greifswald'

call mysubprog hanse. , testvar

-- After return from mysubprog the changes it made to
-- stem variable hanse. are visible here:

say hanse.77.88     ⇒    Demonstration     new element created
say testvar         ⇒    Ostsee            NO change to simple variable
say result          ⇒    Rückgabe          set by RETURN in SUBPROG
```

ooRexx program **mainprog.rex** uses its stem variable `hanse.` as an argument when calling program **mysubprog.rex**. Separated by a comma, the name of simple variable `testvar` is transmitted as second argument.

```
-- File mysubprog.rex
 use arg demo. , ordinary

-- The first argument provided by mainprog.rex is its stem
-- variable hanse. which is named demo. in this subprogram.
-- But both names refer to the same object.
say demo.1                              ⇒   Bremen
demo.77.88 = 'Demonstration'

-- On the other hand, changes to a simple variable are
-- not visible to the calling program.
say ordinary                            ⇒  Ostsee
ordinary = 'abcd'

return 'Rückgabe'           -- conventional return of data
```

Keyword USE ARG in **mysubprog** is used to access the supplied arguments. It does not need to know the names used inside the calling program. It simply uses its own. In this example, the name **demo.** is used for the stem (first argument) and **ordinary** for the simple variable (second argument).
All changes, deletions or additions to stem **demo.** are in reality made to stem **hanse.** – when control returns to **mainprog**, it sees the changes made.
On the other hand, *simple* variables are *read only*. The called program sees the value of **testvar** in its own variable **ordinary**. But any changes remain invisible to the calling program.

```
call myprog                 -- calling MYPROG without argument

-- Different treatment when MYPROG expects an argument:

arg aparm                   -- arg   sees a null string
say aparm                   ⇒  ''

use arg bparm               -- use arg   sees an undefined Variable
say bparm                   ⇒  'BPARM'
```

If **ARG** encounters an unused argument, it uses the *null string* as its value. On the other hand, **USE ARG** treats it like an unknown variable (see rules on page 55).
ARG and **USE ARG** can be used in parallel. They read the same argument string.
USE ARG accepts multiple arguments only if separated by **commas**. If no name is coded between two commas, **USE ARG** ignores that part of the argument string. This allows to use the conventional blank separated arguments intended for processing by **ARG**.
Do not try to initialize a stem accessed via **USE ARG** to a default value (page 38). It creates a new private object in the called program with the same name as the original stem. This severs the connection to the stem in the calling program.

12.1 USE ARG and Arrays

According to the documentation, **USE ARG** supports sharing all types of data collections. The following example shows code for sharing an array (see page 44) in place of a stem.

```
-- Code lines in mymainprog.rex

 phonedirectory    = .array~new          -- create array phonedirectory
 phonedirectory[5] = 'this is a string'

call mysubprog phonedirectory

say phonedirectory[5]             ⇒  'changed string'
say result                        ⇒  '0'
```

This template of a calling program shows the creation of an empty array. Then an item at index position 5 is defined. On the subprogram call, the plain array name is used. Nothing to indicate this being an array is needed.

```
-- Code lines in mysubprog.rex
use arg verzeichnis                    -- local name of array

if verzeichnis~isA(.array) then nop    -- optional: test for array
else return -1                         -- not an array

say verzeichnis[5]                     ⇒  'this is a string'

verzeichnis[5] = 'changed string'
return 0                               -- return value 0 to caller
```

In the called program, the argument is accessed using name **verzeichnis**. Due to the sharing of control information, ooRexx knows it is an array. The **~isA** method can be used by the application to test for the correct object type.[1] The contents of item 5 is displayed and then changed.
The sharing of the array follows the same rules as described for the stem variable previously. All changes to array **verzeichnis** by the subprogram do really happen in array **phonedirectory** of the calling program. Both names reference the same array object.

12.2 Template for Your Own Library of Functions

Usually, an ooRexx program is started using the file name. Over time, more and more useful small tools are written. This creates a growing number of small program files. To consolidate a number of programs –written in ooRexx– into a single library, directive **::ROUTINE** is the tool of choice.

```
-- File mylibrary.rex as template for creating a ooRexx program library:

-- Rexx program code preceding the first ::DIRECTIVE is called prologue
-- In this example, no prologue is used.

-- The first directive here is used to set NUMERIC DIGITS for all routines.
::OPTIONS digits 16                -- available since ooRexx 4.0

::ROUTINE cylinder PUBLIC          -- 1. subprogram  cylinder  starts here
use arg arg1 , arg2 ...            -- read arguments with  ARG and/or USE ARG
...                                -- Rexx code to do the calculations
return volume surfacearea ...      -- result string may contain blanks

::ROUTINE cone     PUBLIC          -- 2. subprogram  cone
use arg arg1 , arg2 ...            -- as above
...
number = uprog(abc)                -- call of a private subprogram (see below)
...
return volume surfacearea ...      -- end of cone main code
uprog:                             -- label starts a private subprogram uprog
...                                -- which is only visible to cone
return something                   -- return of uprog  result to cone

::ROUTINE pyramid  PUBLIC          -- 3. subprogram pyramid
use arg arg1 , arg2 ...
...
return volume surfacearea ...

::ROUTINE name     PUBLIC          -- and so on ...
use arg arg1 , arg2 ...
...
return stringdata
```

Regarding the „prologue“ see page 19. Each **subprogram** in this library begins with a ::ROUTINE directive and is ended by the next ::ROUTINE or other directive (or by end of file). Option PUBLIC is needed to make it callable from outside the library. An important property is the **isolation** of the routines from each other. Everything between two ::ROUTINE statements is invisible to the others. This also applies if a routine calls another in the same library.
Internal subprograms used by a routine are identified by a label, like the example **uprog:** in routine **cone**. All variables defined in **cone** are visible to and changeable by the internal subroutine **uprog**.

[1] For simplicity of the example, an appropriate error handling is not shown.

This has always been Rexx standard for label calls within the same file. Keywords PROCEDURE and EXPOSE (beyond the scope of this Short Reference) are available to control this.
Internal subprograms of a routine are invisible to the other routines in the library. In the example, **uprog** can only be called by **cone**.

```
-- Example of using routine PYRAMID residing
-- in program library MYLIBRARY.REX
...

call pyramid argument1 , argument2 ...

...

-- To use PYRAMID, the calling program
-- needs the following directive appended to it:
::REQUIRES mylibrary.rex
```

The above example of calling PYRAMID shows using keyword CALL. Alternatively, it is also posible to use function syntax:

```
result = pyramid(argument1,argument2)
```

In any case, the program must be told which library contains PYRAMID. This is done by adding a **::REQUIRES** directive at the end of the program file.
Extension **.rex** may be omitted. Since February 2020, ooRexx automatically searches for a file with extension **.cls** and then **.rex**.

13 Classic Style Versus Object Oriented

This chapter compares a classic style solution of *sorting an array* with the object oriented way of coding. Both use method **~sortwith**.
The data to be sorted is in file `hanse.dat` with the following layout:

```
0421  Bremen
040   Hamburg
0451  Lübeck
03841 Wismar
0381  Rostock
03831 Stralsund
03834 Greifswald
```

Area codes start in column 1 and are, as can bee seen, up to 5 characters long. The town names start in column 7 and the longest has 10 characters. Both are left adjusted.

13.1 Using SORTWITH in a Classic Program

When starting the program, the desired sort (by area code or town name) is the only argument:

```
arg a1 .                    -- Sort: A by area code or T by town
select case a1
when 'A' then do
     start = 1
     len   = 5
     end
when 'T' then do
     start = 7
     len   = 10
     end
otherwise
     say 'This program expects A or T as argument.'
     exit 24
     end
```

Dependent on argument **A**[rea code] or **T**[own name], the sort columns are assigned. Town names shorter that 10 characters pose no problem.

```
infile  = .stream~new{'hanse.dat'}    -- the file stream to read ...
tabelle = infile~arrayin              -- Array TABELLE receives
```

Method `arrayin` creates array `tabelle` and copies the file lines to it.

```
tabelle~sortwith(.ColumnComparator~new(start,len))
```

This is the sort instruction. First an object of class `ColumnComparator` is created, which will use the columns defined in variables *start* und *len*. This object is then used by method `SortWith` to perform sorting of array `tabelle`.

```
do idx over tabelle
     say idx
     end
```

To write the result to the screen using the `say` keyword, a `do ... over` loop goes through all used items of array `tabelle`. In place of `idx` any convenient variable name could be used.

```
0381  Rostock
03831 Stralsund
03834 Greifswald
03841 Wismar
040   Hamburg
0421  Bremen
0451  Lübeck
```

This is the screen output of the above `do ... over` loop.

```
-- alternative: conventional loop
do i=1 for tabelle~items
     say tabelle[i]
     end
```

Altenatively this conventional loop with an iteration variable could be used. Each item is accessed using the [] syntax (without period, because this is an array, not a stem variable). Method `items` returns the array size. Because we know there are no unused items, this is the number of loop iterations.
This concludes the classic style example.

Other Sort Sequence

```
tabelle~sortwith(.CaselessColumnComparator~new(start,lenght))
```

If upper- and lowercase letters are to be treated as equal, the `Caseless` version of the sorting class is used. As already mentioned, this recognizes only the 26 letters of the English alphabet.

```
-- To invert sorting result use:
--            .InvertingComparator~new( ── myComparatorClass~new(args) ── )

tabelle~sortwith(.InvertingComparator~new(.ColumnComparator~new(start,length)))
```

Class `InvertingComparator` changes the sorting sequence from ascending to descending. It expects the name of the class which will do the actual sort as argument of its `new` method.

13.2 Object Oriented Use of SORTWITH

This section[1] is expressly not intended as a "how to". It is limited by what I learned when trying out object oriented code.
First, a number of object names have to be defined. Each line in `hanse.dat` contains 2 fields (attributes). They will be called `areacod` and `townname` here. We also need a name for the resulting records structure and choose `hansesort` for this.

Setting Up a Class File

To this end, we create a fittingly named file `oohanse.cls` :

```
::class hansesort public inherit comparable
```

This defines class `hansesort` as being useable by any program (`public`). Because its purpose is sorting, it inherits the properties of predefined class `comparable`.

```
::attribute areacod
::attribute townname

::method init
  expose         areacod townname
  use strict arg areacod, townname
```

[1] It is derived from ooRexx sample program `sortComposite.rex`.

The ::ATTRIBUTE directives define the data fields in structure `hansesort`. This automatically triggers in the background the creation of a method with the same name as the attribute for read and write access.
When defining a data object, a method with the the prescribed name `init` must be defined. Use of keyword `expose` is required here –and for all other methods– to define, which fields (attributes) this methode may read or change. All other variables inside the method remain invisible to the outside world. To create a record structure as defined in the `hansesort` class, `init` of course needs to access all its fields (attributes). Therefore, keyword `expose` has a complete, *blank separated* list of the fields defined through ::ATTRIBUTE directives.
When called, a method normally receives arguments. Instruction `use strict arg` defines how the arguments are parsed into attributes. Again, for our sort we need a list of all defined attributes. This time a *comma separated* list is required. Option `strict` ensures that the correct number of arguments is present.

```
::method string                     -- method STRING of class HANSESORT
  expose         areacod townname        -- converts objects ...
  return '>'areacod' -- 'townname'<'     -- into a string for SAY
```

The record format (the class) `hansesort` is a structure, not a simple character string. If a structure is encountered by string oriented processes, like keyword **say**, ooRexx uses a default method to convert the data into a chracter string. The alternative is to define a method named `string` for the class. Here a very trivial example is used which inserts some special characters to make obvious that method `string` of class hansesort is active.

Classes for Sorting

For each sorting function (by area code and by town name) a subclass of predefined class `comparator` must be defined. Each class needs a method with the predefined name `compare`which controls the sort.

```
::class AREAsorting public subclass comparator
::method compare
use strict arg lllll, rrrrr
return lllll~areacod~compareto(rrrrr~areacod)
```

Class `AREAsorting` sorts by contents of field (attribute) `areacod`. Method `compare` receives as arguments two consecutive values from field `areacod` übergeben. For this example, the variables `lllll` and `rrrrr` are used. Using the builtin comparison method `compareto` the rather involved expression after `return` returns value 1, 0 or -1. This way method `compare` tells ooRexx whether relation *larger*, *equal* oder *smaller* applies to comparing pair `lllll` and `rrrrr` applies.

```
::class TOWNsorting public subclass comparator
::method compare
use strict arg lllll, rrrrr
return lllll~townname~compareto(rrrrr~townname)
```

Class `TOWNsorting` functions the same way, but uses field (attribute) `townname`. For any additional sorting field, a corresponding `comparator` subclass with its method `compare` is needed.

```
return -lllll~townname~compareto(rrrrr~townname)
```

This line is **not** present in the class file. It shows how to invert the sort order. The minus sign at the beginning of the expression after `return` negates the results (-1, 0, 1) to (1, 0, -1).
This way, two additional classes could be easily coded for *descending* sort by area code or town name.
This completes class file `oohanse.cls` and we can code the program to use it.

ooRexx Code Using the Sorting Classes

The program file is named `oohanse.rex`:

```
myfile  = 'hanse.dat'                   -- name of data file
mytable = .array~new                    -- create empty array MYTABLE
```

These two lines are the same as in the classic example of page 51.

```
do i=1 while lines(myfile)
     myline = linein(myfile)                          -- read line and ...
     parse var myline 1 afield 7 tfield               -- parse into 2 fields
     mytable~append(.hansesort~new(afield,tfield))    -- uses class HANSESORT
     end i
```

Writing the data to array `mytable` is done using class `hansesort` to create objects as defined. Because this class expects 2 arguments, each file record has to be parsed into 2 fields (attributes). Number and sequence of attributes in the arguments passed to methode `new` must be identical to the actually used method `init` of class `hansesort`. In place of variables `afield` and `tfield` any names may be used.

```
arg a1 .                -- A or T for sorting by area code or town
select case a1
when 'A' then mytable~sortwith(.AREAsorting~new)
when 'T' then mytable~sortwith(.TOWNsorting~new)
otherwise
     say 'This program expects A or T as argument.'
     exit 24
     end
```

This is the sorting step. Depending on using argument A or T when starting the program, the appropriate subclass is used by method `sortWith`.

```
do idx over mytable
     say idx       -- SAY implicitly uses the STRING method of HANSESORT
     end
```

The loop to write the result to the screen is the same as in the classic example.

```
::REQUIRES oohanse.cls
```

The ::REQUIRES directive at the end of the program tells ooRexx where to find the class definitions. Since February 2020 ooRexx automatically searches for files with extension `.cls` if none is given.
An alternative would be to copy the complete file `oohanse.cls` in place of the ::REQUIRES directive into the program file. In this case the the `public` options in the class definitions were not required. And it would make the class definitions inaccessible to other programs.

```
>0381  -- Rostock<
>03831 -- Stralsund<
>03834 -- Greifswald<
>03841 -- Wismar<
>040   -- Hamburg<
>0421  -- Bremen<
>0451  -- Lübeck<
```

Independent of placing the class definitions, the screen output –sorted by area code– will be as shown above.
The items in array `tabelle` are now objects as defined in class `hansesort`. Our method `string` has done the preprocessing for keyword SAY as the additional special characters show.

Comparison of Effort

Documenting the classic solution in section 13.1 uses 265 mm of vertical text area. Its object oriented alternative in section 13.2 uses 515 mm, or 1.94 times the space.

14 Some ooRexx Fundamentals

14.1 How Code Lines Are Processed

Everything not included in quotes (simple or double) is folded to UPPERCASE before it is interpreted. Text **wordpos** as well as **WordPos** is seen as **WORDPOS** by ooRexx. Mixed case solely improves readability for the human eye.
The steps of interpreting a code line are as follows:

```
-- The following variable exists:
timezone = 'Daylight Saving Time'

-- Line of code in the program file:
say 'It is'   now      time() "hours   "     timezone

-- 1. All letters outside of quotes are folded to UPPERCASE
--    und multiple blanks are reduced to 1 blank:
SAY 'It is' NOW TIME() "hours   " TIMEZONE

-- 2. Function calls and variables are replaced by their (return-)values:
SAY 'It is' NOW 15:19:54 "hours   " Daylight Saving Time

-- 3. Keyword instructions are executed. SAY writes to the screen:
It is NOW 15:19:54 hours    Daylight Saving Time
```

Step by step:

- **SAY** is recognized as an ooRexx keyword. It writes data to the screen.
- Character string **It is** remains unchanged, because it is included in single quotes.
- **NOW** is *not* recognized as a variable or something else with a meaning to ooRexx. It remains in place while the number of blanks, separating it from its neighbours, is reduced to 1.
- **TIME()** is the name of a builtin function. The call is replaced by the character string returned by this function.
- Character string **hours** and the three blanks following it are enclosed in double quotes and remain unchanged.
- **TIMEZONE** is recognized as a variable. It is replaced by its value **Daylight Saving Time**. Again, the number of blanks separating it from its neighbours, is reduced to 1.
- Consequently, in the output, word **hours** is followed by 4 blanks: 3 inside the quoted string, plus the single separator blank.

Concatenation

```
filename  = 'doreadme'
extenta   = 'txt'
extentb   = '.dat'

say filepath'.abc'         ⇒    doreadme.abc      -- variable followed by string
say filepath'.'extenta     ⇒    doreadme.txt      -- string between variables
say filepath||extentb      ⇒    doreadme.dat      -- two variables
say 'DAT'random()extentb   ⇒    DAT152.dat        -- string, function call, variable
```

If blanks are unwanted, quoted character strings and variables may touch each other. The same applies to quoted strings and function calls. To concatenate the values of two or more variables or appending a function call to a variable, the operator || is used.
A variable name which immediately follows a quoted string, should not be **x** or **b** to prevent ooRexx from confusing it with a hex- or binary string (see page 33).

Line Continuation

```
say 'This text' ,
    'is logically placed in a single line.'

say 'This is' ; say 'logically divided' ;  say 'into 3 lines.'
```

A *comma* as the last character of a line is always interpreted as a continuation character. It may appear at any position where a blank is allowed. The resulting logical line has a blank at the comma position. A *semicolon* has the opposite effect and acts as an end of line character. Everything following the semicolon is considered a new line by ooRexx.

Comments

```
/* REXX on mainframe, OS/2 and DOS required programs to start with a comment line */
/*
   a comment may span          /* comment within comment is possible, if complete */
   multiple lines
*/
say 'Today is' date() /* in the middle of a code line */ time()
-- (double hyphen) is a line comment operator introduced with ooRexx 3.0
say 'Heute ist der' date()             -- makes the rest of the line a comment
```

The sample programs of the ooRexx installation have the string `#!/usr/bin/env rexx` as first line. This is an instruction for running on a Unix system. It is ignored on Windows.

Ordinary and Strict Comparison Operators

Conditional terms using these operators[1] compare return the comparison result either as 0 for *false* or 1 for *true*. They are used with keywords **if, when, while, until**.

```
comparison operators          strict comparison operators
<  <=   =   >=  >  \=           << <<=  == >>=  >> \==
```

If both terms are valid numbers, [2] a numeric comparison is done. Otherwise a string comparison is done. Leading blanks[3] are ignored. The shorter string is padded on the right with blanks.
In a **strict** comparison, byte by byte is examined. No padding is done. Two strings of different length are not equal; 1 and 1.0 are not equal. The keyword **select case**, introduced in ooRexx 5.0, always uses strict **==** comparisons on the **when** instructions (see example on page 51).
Comparisons may be logically combined using & (AND), | (OR) and && (XOR). In any case, *all* comparisons of a conditional statement are processed.

```
if datatype( result ,'N') , result > 0 , result < 100 then ...
```

Since ooRexx 3.2, the *comma* as conditional AND can be used. The leftmost term is processed first. Only if it returns 1, the next comparison is processed. Result 0 skips all remaining terms and returns 0.
For example, a test for a valid number could be first. If it fails, runtime errors –triggered if a following term would encounter a non-numerical value– are avoided.

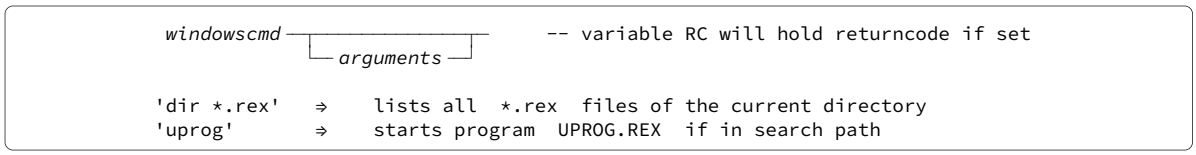
```
if lines(myfile) = 1 then ...      -- equivalent:   if  lines(myfile) then ...
if lines(myfile) = 0 then ...      -- equivalent:   if \lines(myfile) then ...
```

If a function returns –or a variable has a value of– either 0 or 1, a = comparison need not be coded after **if, when, while, until**. Character \ is the logical *negation* operator,. reversing 0 to 1 and 1 to 0.

[1] In addition to the shown 12 operators, another 12 formats with identical function exist. They are not shown here.
[2] Be careful. Some hexadecimal strings, for example 0E01, are treated as valid numbers by ooRexx, because: $0E01 = 0 \cdot 10^1 = 0$. Therefore, hexadecimal strings should be compared by a strict operator.
[3] Actually, also tab characters; see **xrange** BLANK on page 9.

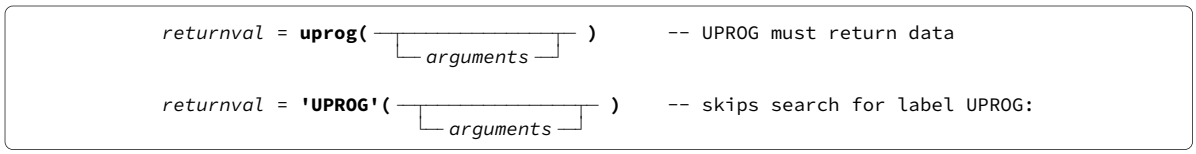
14.2 Branching to Subprograms

Processing as if Entered in the Command Line



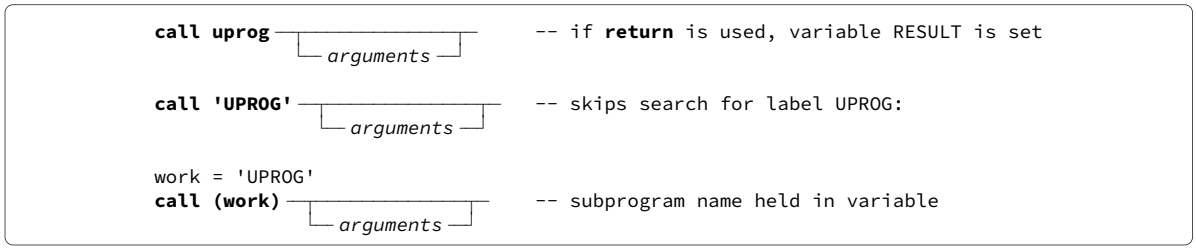
If ooRexx, after interpreting a line of code, encounters an unrecognized string at the beginning of the line, this line is considered to be a Windows command and is passed on to the operating system for execution. Windows treats it as if entered on the command line.
Windows uses the search order: system command, .EXE file, .BAT file, .CMD file and finally .REX file. If nothing is found, an error message appears *and the program continues*.
An ooRexx program may use the keyword **exit** to report back a number to the calling program. This sets special variable **RC** (return code) in the calling ooRexx program. If nothing follows **exit**, RC is set to 0, meaning no error. Otherwise, the number signals the type of error. A selection of return codes as used by Windows is shown on page 59.

Function Call



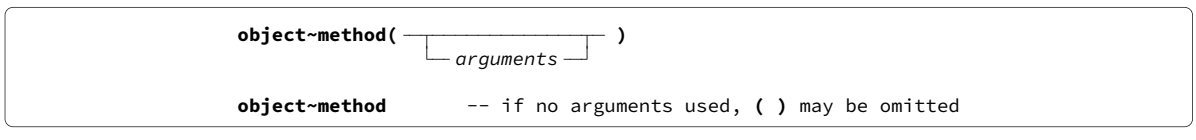
A function type of call is recognized by a name being *immediately* followed by a **(**. Commas are used to logically separate *argument strings* from each other. See the syntax example on page 60 (top).
To be callable as function, the subprogram **must** use the keyword **return** to return a string –which my be null– to the calling program. Otherwise a runtime error stops the program.
In the calling program, the function is replaced by the returned string. Therefore, the line **must** have code that processes the returned data. For example, the function call is on the right hand side of an assignment statement. Otherwise, the returned data, unrecognized by ooRexx, will be handed over to Windows as a command string.

Keyword CALL



Using keyword **call**, the called subprogram need not return a string. If it does, keyword **return** is used. The calling program finds this data in special variable **RESULT**. The comma may be used as argument separator. Make sure to code something after the last comma, to prevent it from being treated as line continuation.
Only the keyword **call** offers a way to use a variable holding the subprogram name.

Method Call



A method is identified by a tilde ~ appending it to the object it works on (its first argument). Additional arguments, if used, are coded in function syntax. From a classic coding point of view, methods show a combination of the previous call types:

- In common with function calls, method calls are replaced in the code by the data they return.
- Quite contrary to function calls, no harm is done if a *method* call is alone on a code line. Any returned „orphan“ data is ignored by ooRexx. But in common with CALL, the returned data is then available as variable **RESULT**.
- The **( )** may be omitted if no arguments are used.

The search sequence depends on the currently active hierarchy of classes.

14.3 ooRexx Search Sequence for Subprograms

A subprogram to be called may be a part of ooRexx, an external program written in ooRexx or residing in a library. Apart from the condition that functions *must* return data, there is a free choice of using a function call or the CALL syntax. In both cases:

```
uprog()
call uprog
```

the search sequence is identical.[4] As soon as the first of the following condition is true, processing branches to that code:

- Does a label **uprog:** exist in the current program file? This step is skipped, if the name is included in single or double quotes.
- Is **uprog** a builtin part of ooRexx? Which are:
  - functions implemented in the interpreter[5] and
  - the **Rx...** and **Sys...** functions from library **rexxutil.dll** that comes with ooRexx.[6]
- Does a directive **::ROUTINE uprog** exist in the current program file?
- Exists, in one of the files listed in ::REQUIRES directives, a directive **::ROUTINE uprog PUBLIC**?
- Does in *Rexx Macrospace*, among the files loaded with **Before** option, a program named **uprog** exist?
- Does name **uprog** exist in an *external function library* (DLL file) that was loaded via directive **::REQUIRES *name* LIBRARY**?
- Does a file named **uprog.rex** exist ...
  - in the current directory, or
  - in a directory of the PATH environment variable – which by default includes the ooRexx installation directory?
- Does in *Rexx Macrospace*, among the files loaded with **After** option, a program named **uprog** exist?
- ooRexx stops and reports: *Error 43: Could not find Routine UPROG*

What is a Rexx Macrospace?

Rexx programs can be permanently loaded into RAM storage by the operating system. This saves disk access operations. The necessary functions (SysAddMacroSpace and others) are part of the Rexx Utilities.
This method has its origins on the mainframe, where, known as *Nucleus Extension*, it was an extremely effective speed-up. On the other hand, ooRexx and Notebooks nowadays have become very fast even without this trick. Therefore I have never tried it in practice.

[4] For better redability the internal folding of names to uppercase is not shown here.
[5] Chapter 7.4 *„Built-in Functions“* in file *rexxref.pdf*
[6] Chapter 8 *„Rexx Utilities“* in file *rexxref.pdf*

What is an External Function Package?

ooRexx offers two application programming interfaces to programs written in C (older) or C++ (current) and residing in a DLL. These then can be called like any other function to process data. To include a DLL in the search, directive **::REQUIRES** ***name*** **LIBRARY** is used. An example is **rxmath.dll** (page 18).

What is BSF4ooRexx?

This is the name of a tool developed since ooRexx 4.1 by professor *Rony G. Flatscher* of Wirtschaftsuniversität Wien, and many contributors, which provides a very powerful interface to the **Java** environment in both directions. It makes everything available in Java, for example a graphical user interface, callable from ooRexx programs.
The most current BSF4ooRexx version can be found at link:
`sourceforge.net/projects/bsf4oorexx/files/GA`

14.4 Some Windows Returncodes

```
SysGetErrorText( — rc — )

-- SysGetErrorText(5)   ⇒    Access denied
```

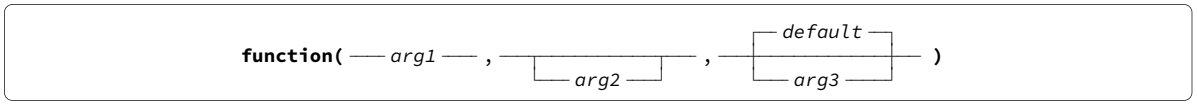
If Windows cannot serve a request, the reason is reported using a numerical return code. The above function returns a short message, indicating the reason of the failure. A selection of some error codes and texts follows:

```
     2  File specified not found
     3  Path specified not found
     4  Cannot open the file
     5  Access denied
    13  Invalid data
 8, 14  Not enough storage for the command/operation
    15  Cannot find the specified drive
    16  Directory cannot be removed
    17  System cannot move file to a different disk drive
    18  No more files
    19  Write protected media
    23  Data error (CRC-check)
    26  Specified disk cannot be accessed
    32  No access, file is being used by other process
    36  Too many files opened
    39  Disk is full
   183  A file with this name already exists
```

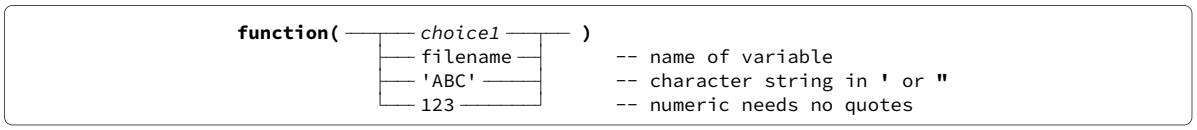
ooRexx in case of errors uses its own, different return codes. The texts can be obtained the same way, using function **ErrorText()**. Appendix C of *rexxref.pdf*) contains an extensive printed list.

15 How to Read the Syntax Diagrams

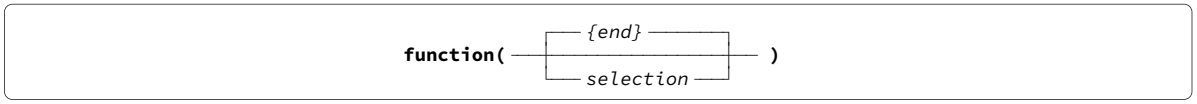
In the diagrams, an *italic* font represents *variables, numbers, character strings*. The latter are included by double " or single ' quotes. Numbers need no quotes.



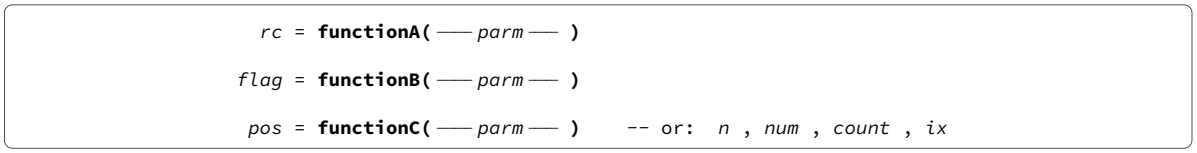
This represents a function that expects 3 comma separated *arguments* (also called parameters). In this example, *arg1* must be filled in, while the other two may be omitted.
The value **above** the main line shows what is used as *default* if the 3rd argument is omitted.
For clarity, any separating **commas** are always shown. As a rule, all commas *to the right* of the last used argument may be omitted.



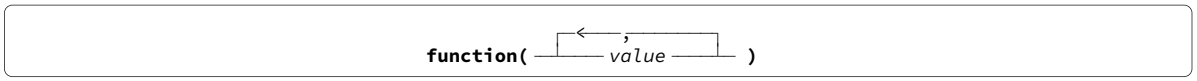
A „ladder“ shows a selection of possible arguments. One must be used; there is no default.
A quoted string may be used in place of a variable if its simpler or more readable. The quotes make sure that the character string ABC is used, even if a variable ABC with an unknown value exists. Particularly with short strings, overlooked variables are a typical source of unexpected behaviour.



In several cases the actual default depends on the data and is not a definite value. Then a short symbolic word included in *{ }* is used. For example *{end}* stands for „to the end of the string“.



Assignment statement layout is used for function and method syntax diagrams, when it is important to indicate the kind of returned data, for example:
A return code *rc* will be 0 if no error occurred. Any other number indicates the type of error.
If *flag* represents the returned data, the possible values are 1 for **true** or 0 for **false**. In some cases -1 is possible.
A return name of *pos*, *n*, *num*, *count* or *ix* indicates a positive whole number or zero, representing a length, counter, number or an array index location (the latter never 0).



This function accepts more than one argument string, separated by commas.

Index

Contents