# 

**Octave**

*a programming language for scientific computing*

[**Concepts**](#_jz0y5hyzut62) **3**

[**Variable**](#_xmvx9bw5cula) **3**

[**Operations**](#_gwcjlp8jtx8j) **4**

[**Data Types**](#_oncpv9wtbvts) **5**

[Numeric](#_5eo7cg78c2i2) 5

[String](#_5r1w8d51rmmj) 6

[**Objects size**](#_k3wewwaj6we7) **9**

[**Containers**](#_v2jaa3y9y2bh) **10**

[Structures](#_p2dype88qirl) 10

[Containers.Map](#_89q1y7xo6oy) 11

[Cell Arrays](#_2zl3c4erssln) 11

[**Arrays**](#_68qff2m43qax) **12**

[Vector](#_hm7cjhj1qh3s) 12

[Matrix](#_9onbnfvvm9y5) 12

[**Statements**](#_jshi4ek34zyy) **12**

[FOR](#_imtkockcxd8d) 12

[WHILE](#_upnqgw45i73n) 12

[DO-UNTIL](#_2tr7q5km1ein) 13

[IF-ELSE](#_jbl69udorcym) 13

[SWITCH](#_r2qbkzxg3l9w) 13

[TRY-CATCH](#_tm56gehxf62o) 13

[**Functions**](#_y8goutrzyrfk) **13**

[**Data Manipulation**](#_ofzbh54f2pmv) **14**

[**Data**](#_ah3my8j3xhd8) **14**

[**Graphics**](#_7b32nh6vgdnf) **14**

# 

# 

# Concepts

Is a high-level programming language primarily intended for scientific computing and numerical computation.

Helps in solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB.

Adding *#! Full-Path-Octave-Binary* in the first row of the script and make the file executable *chmod command*, you can simply type: *Script-name* on the terminal and the script will be executed.

Doc: https://docs.octave.org/latest/index.html#SEC\_Contents

# Variable

* 1. Assignment *- Dynamic Typing*

global a = 2; % make a global var

persistent a = 2; % make a persistent var

b = “string”; *% better use “ - used only for this*

[c, d] = 1, ‘string’;

* 1. Continuation Lines *- ignore \n*

x = long\_variable\_name ... *% comment one*

+ longer\_variable\_name … *% comment two*

- 42 *% last comment*

* 1. Print

disp(a);

disp(sprintf('2 decimais %0.2f',a)

* 1. Format

format long %14 decimal places

format short %4 decimal places

# Operations

Comment

%

Math

: plus (x, y, ...) is equivalent to x **+** y

: minus (x, y) is equivalent to x **-** y

: mtimes (x, y, ...) is equivalent to x **\*** y

: mrdivide (x, y) is equivalent to x **/** y  *- right division*

: mldivide (x, y) is equivalent to x **\** y *- left division*

: mpower (x, y) is equivalent to x **^** y

: times (x, y, ...) is equivalent to x **.\*** y *- element-by-element*

: rdivide (x, y) is equivalent to x **./** y *- element-by-element*

: ldivide (x, y) is equivalent to x **.\** y *- element-by-element*

: power (x, y) is equivalent to x **.^** y *- element-by-element*

: transpose (x) is equivalent to x**.'**

: ctranspose (x) is equivalent to x**'**  *- Complex Conjugate Transpose*

: uminus (x) is equivalent to **-**x

: uplus (x) is equivalent to **+**x

Logical

: isequal (x1, x2, …) *is equivalent to* x1**==**x2**==**...

: eq (x, y) *is equivalent to* x **==** y

: ge (x, y) *is equivalent to*  x **>=** y

: gt (x, y) *is equivalent to* x **>** y

: le (x, y) *is equivalent to*  x **<=** y

: lt (x, y) *is equivalent to* x **<** y

: ne (x, y) *is equivalent to*  x **!=** y *and* x **~=** y

Decision

: and (x1, x2, …) *is equivalent to* x1 **&** x2 **&** … *% element-wise*

: or (x1, x2, …) *is equivalent to* x1 **|** x2 **|** … *% element-wise*

: not (x) *is equivalent to* **!**x *and* **~**x

: xor (x, y) - *exclusive or*

**&&** **||** *short-circuit boolean ops (and - or)*

Boolean

ans = 0 -> False ans = 1 -> True

Incremental

+= -= \*= /= *% by a value*

++x --x *% by one*

# Data Types

Return the **type** of the expression *expr*, as a string. If *expr* is omitted, return a cell array of strings containing all the currently installed data types.

: typeinfo ()

: typeinfo (expr)

Return the **class** of the object *obj*, or create a class with fields from structure *s* and name *id*. Additional arguments name a list of parent classes from which the new class is derived.

: class (obj)

: class (s, id)

: class (s, id, p, …)

**Convert** *val* to data type *type*. The input *val* may be a scalar, vector, or matrix of a class that is convertible to the target class. If a variable *var* is specified after *"like"*, *val* is converted to the same data type and sparsity attribute. If *var* is complex, *val* will be complex, too.

: cast (val, "type")

: cast (val, "like", var)

## Numeric

May be a **scalar**, a **vector**, or a **matrix**, and it may contain **complex** **values**. Hexadecimal notation starts with *‘0x’* or *‘0X’*, binary notation starts with *‘0b’* or *‘0B’*, otherwise decimal notation is assumed.

Can use *underscore* to separate the numbers, but the Octave interpreter will ignore it.

42 *% decimal notation* 0b101010 *% binary notation*

0x2A *% hexadecimal notation* 0b10\_1010 *% underscore notation*

In **decimal notation**, the numeric constant may be denoted as decimal fraction or even in scientific (exponential) notation. Not possible for hexadecimal or binary notation.

.105 = 1.05e-1 = .00105e+2

**Complex numeric** constants are denoted as the sum of real and imaginary parts. The *imaginary* *part* is denoted by a *real-valued numeric constant* followed immediately by a *complex* *value* indicator (‘i’, ‘j’, ‘I’, or ‘J’ ) without space between them.

3 + 42i = 3 + 42j = 3 + 42I = 3 + 42J = 3.0 + 42.0i

3.0 + 0x2Ai = 3.0 + 0b10\_1010i = 0.3e1 + 420e-1i

**Convert** *x* to *double* - *8-bit integer* - *unsigned 8-bit integer* precision type. There is also for 16-bit, 32-bit and 64-bit

: double (x)

: int8 (x)

: uint8 (x)

**Return a complex value from real arguments.** With 1 real argument x, return the complex result x + 0i. With 2 real arguments, return the complex result re + imi.complex. complex (x)

: complex (re, im)

To **check** some **types**:

: isnumeric (x) - : islogical (x) - : isbool (x)

: isfloat (x) - : isreal (x) - : iscomplex (x)

: ismatrix (x) - : isvector (x) - : isrow (x) - : iscolumn (x)

: isscalar (x) - : issquare (x) % a 2-d square array

## String

A string constant consists of a sequence of characters enclosed in either double-quote or single-quote marks. Expect them to be UTF-8 encoded Unicode strings. Internally the string is actually a row vector of length N and containing the ASCII code for every letter.

In a character matrix, each row (can be a col) is a string, and the column is the character in position c. If one of the words is smaller than others, Octave will put blank characters at the end of them.

Return true if x is a character/string array.

: ischar (x)

: isstring (x)

**Concatenate** some string.

["str1", "str2", ..., "strN"] -> "str1str2...strN"

Return a string of n blanks.

: blanks (n)

**Transforme** all letters to **lower**/**upper** **case**.

: tolower (s) - : lower (s)

: toupper (s) - : upper (s)

**Remove** *trailing whitespace and nulls* from s. If s is a matrix, deblank trims each row to the length of the longest string. If s is a cell array of strings, operate recursively on each string element.

: deblank (s)

**Remove** *leading and trailing whitespace* from s. If s is a matrix, strtrim trims each row to the length of the longest string. If s is a cell array of strings, operate recursively on each string element.

: strtrim (s)

**Truncate** the character string s to length n. If s is a character matrix, then the number of columns is adjusted. If s is a cell array of strings, then the operation is performed on each cell element and the new cell array is returned.

: strtrunc (s, n)

Return the **substring** of s which starts at character number offset and is len characters long. Position numbering for offsets begins with 1. If the offset is negative, extraction starts that far from the end of the string.

If len is omitted, the substring extends to the end of s. A negative value for len extracts to within len characters of the end of the string.

: substr (s, offset)

: substr (s, offset, len)

**Split** the string str using the delimiters specified by del and return a cell string array of substrings. If a delimiter is not specified the string is split at whitespace {" ", "\f", "\n", "\r", "\t", "\v"}. Otherwise, the delimiter, del must be a string or cell array of strings.

: [cstr] = strsplit (str)

: [cstr] = strsplit (str, del)

: [cstr, matches] = strsplit (…)

*Valid for the two below:* If either s1 or s2 is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Return **1** if the character strings s1 and s2 are the **same**, or **0** otherwise.

: strcmp (s1, s2) *% case sensitive*

: strcmpi (s1, s2) *% not case sensitive*

<,,,,,,

Return **1** if the first n characters of str s1 and s2 are the **same**, or **0** otherwise.

: strncmp (s1, s2, n) *% case sensitive*

: strncmpi (s1, s2, n) *% not case sensitive*

**Check** whether string(s) **start**/**ends** with **pattern**(s).

Return an array of logical values that indicates which string(s) in the input str (a single string or cell array of strings) end with the input pattern (a single string or cell array of strings). If the value of the parameter "IgnoreCase" is true, then the function will ignore the letter case of str and pattern.

: retval = startsWith (str, pattern)

: retval = startsWith (str, pattern, "IgnoreCase", ignore\_case)

: retval = endsWith (str, pattern)

: retval = endsWith (str, pattern, "IgnoreCase", ignore\_case)

**Search** for **pattern** in the string str and return the starting index of every such occurrence in the vector idx.

If there is no such occurrence, or if pattern is longer than str, or if pattern itself is empty, then idx is the empty array [].

The optional argument "overlaps" determines whether the pattern can match at every position in str (true), or only for unique occurrences of the complete pattern (false). The default is true.

If a cell array of strings cellstr is specified then idx is a cell array of vectors, as specified above.The optional argument "forcecelloutput" forces idx to be returned as a cell array of vectors. The default is false.

: idx = strfind (str, pattern)

: idx = strfind (cellstr, pattern)

: idx = strfind (…, "overlaps", val)

: idx = strfind (…, "forcecelloutput", val)

**Replace all occurrences** of the **pattern** ptn in the string str with the string rep and return the result.

: newstr = strrep (str, ptn, rep)

: newstr = strrep (cellstr, ptn, rep)

: newstr = strrep (…, "overlaps", val)

**Delete all occurrences** of **pattern** within str.

: newstr = erase (str, ptn)

**RegEx string matching.** Search for pattern in UTF-8 encoded str and return the positions and substrings of any matches, or empty values if there are n.

: [s, e, te, m, t, nm, sp] = regexp (str, pat)

: […] = regexp (str, pat, "opt1", …)

# Objects size

Return the number of **dimensions** of *a*, as well as the number of **columns** and **rows**.

: ndims (a)

: columns (a) % equivalent to size (a, 2)

: rows (a) % equivalent to size (a, 1)

Return the **length** of the object *a*. The length is 0 for empty objects, 1 for scalars, and the number of elements for vectors. For matrix or N-dimensional objects, the length is the number of elements along the largest dimension.

: length (a)

Return a row vector with the **size** *(number of elements)* of each dimension for the object *a*. When given a second argument, *dim*, return the size of the corresponding dimension, if *dim* is a vector, return each of the corresponding dimensions *- can have multiples dim*.

: size (a)

: size (a, dim)

: size (a, d1, d2, …)

: [rows, cols, …, dim\_N\_sz] = size (…)

Return true if a is an **empty** *(any one of its dimensions is zero)* or **null** **matrix**.

: isempty (a)

: isnull (x)

Return the **size** of val in **bytes**.

: sizeof (val)

Return true if the dimensions of **all arguments agree**.

: size\_equal (a, b, …)

# Containers

Allow different data types in the same variable.

## Structures

The elements of a structure can be of any type, including other structures.

The ‘.’ character separates the structure name (in the example above x) from the field name and indicates to Octave that this variable is a structure.

The simplest way to process data in a structure is within a for loop. Or, you can convert it to another type of container, to a data cell, use: struct2cell(s).

When each field of the structure is a cell array - all have the same dimensions - it’s called Structure Array.

x(1).a = "string1";

x(2).a = "string2";

x(1).b = 1;

x(2).b = 2;

3 ways to create:

(1) x.a = 1;

x.b = [1, 2; 3, 4];

(2) names = ["Bill"; "Mary"; "John"];

ages = [37; 26; 31];

for i = 1:rows (names)

database.(names(i,:)) = ages(i);

(3) s = struct ("field1", {1, "one"}, "field2", {2, "two"}, "field3", 3);

Return the **number of fields** of the structure *s*.

: numfields (s)

Return a **cell array of strings** with the **names of the fields** in the specified input. The names are the elements of the structure.

: names = fieldnames (struct)

Return true if the x **is a structure** and it **includes** an **element** named *name*.

: isfield (x, "name")

Return a **copy** of the structure *s* with the field member *field* set to the value *val*.

: sout = setfield (s, field, val)

**Get the value** of the field named *field* from a structure or nested structure *s*. If s is a structure array then *sidx* selects an element of the structure array, *field* specifies the field name of the selected element, and *fidx* selects which element of the field (in the case of an array or cell array).

: val = getfield (s, field)

: val = getfield (s, sidx1, field1, fidx1, …)

Return a ***copy*** of the structure (arrays) *s* with the field *f* removed. If *f* is a cell array of strings or a character array, remove each of the named fields.

: sout = rmfield (s, "f")

: sout = rmfield (s, f)

## Containers.Map

**Create an object** that stores a list of key/value pairs.

*Keys* is an array of unique keys for the map *- can be numeric scalars or strings*. Multiple string keys are entered as a cell array of strings. *Vals* is an array of values for the map with the same number of elements as keys.

When called with no *input arguments* a default map is created with strings as the key type and "any" as the value type. The *"UniformValues"* option specifies whether the values of the map must be strictly of the same type.

: m = containers.Map ()

: m = containers.Map (keys, vals)

: m = containers.Map (keys, vals, "UniformValues", is\_uniform)

## Cell Arrays

c = {"1", "2", "3"; "x", "y", "z"; "4", "5", "6"}; % Create a cell array

c = cell (2,2) % Create a 2-by-2 cell with empty matrices

c{1, [1,3]} % Indexing - access the elements (return only col 1 and 3)

c(2,1:2) % Indexing - access a sub array of the cell

c{3} = 3 % Override value

num2cell(A) % Convert a numeric matrix A to a cell array

cellstr (strmat) % Create a cell from a string array - each row become a element

iscellstr (cell) % Return T is every element is a character string

cell2mat (c) % Convert the cell into a matrix

cell2struct (cell, fields) % Convert the cell into a structure

# Arrays

Indentation starts with 1 *- in Cell Array too*.

Vector

vector = [1 2 3] %sep is the space

vector = Start:Step:Stop %step is optional - it’s called *Range*

vector(idx) %element of index idx

vector([idx1:idx2) %return from idx1 to idx2

Matrix

mat = [1 2; 3 4; 5 6] %row sep is ; - column sep is space

I = eye(n) % identity matrix nxn

mat = ones(nRows,nColumns) %matrix with only 1

mat = zero(nRows,nColumns) %matrix with only 0

mat = rand(nRows,nColumns) %matrix with only random num between 0 and 1

mat(r,c) %row r and column c

mat([r1 r2],:) %rows r1 and r2 and all columns

mat = [mat, [COLUMN VECTOR]] %add a new column to the right

mat3 = [mat1 mat2] %concatenate side by side (with ; will be one on top of the other)

# Statements

FOR

for var = range % will run for each value in range

*body*

endfor

for [ val, key ] = structure % will run for each val/key of the structure

*body*

endfor

* can use break and continue

WHILE

*Will run while a condition is true*

while (*condition*)

*body*

endwhile

- can use break and continue

DO-UNTIL

*It’ll run at least one time*

do

*body*

until (*condition*)

- can use break and continue

IF-ELSE

if (*condition*)

*then-body*

elseif (*condition2*)

*elseif-body*

else

*else-body*

endif

SWITCH

switch (*expression*)

case *label*

*command\_list*

case *label*

*command\_list*

…

otherwise

*command\_list*

endswitch

TRY-CATCH

try

*body*

catch err

*cleanup*

end\_try\_catch

# Functions

function (var =) funcName(arg1, ..., argN),

*commands*

end;

* *var =* is used to return the *var*
* is a file, and end with .m, like script files
* more than one independent func in a file, it must be a script file

# Data Manipulation

parameters in *italic* are optionals

* 1. Arithmetic *- work for real and complex scalar (x), vector or matrix arguments - x*

exp (x) %compute e^x for each element of x

expm1 (x) %compute exp (x) - 1 accurately in the neighborhood of zero

log (x) %compute the natural logarithm, ln (x), for each element of x

reallog (x) %return the real-valued natural logarithm of each element of x

log10 (x) %compute the base-10 logarithm of each element of x

log2 (x) %compute the base-2 logarithm of each element of x

pow2 (x) %compute 2 .^ x for each element of x

sqrt (x) %compute the √ of each element of x - if x is negative, a complex result is returned

realsqrt (x) %return real-valued √ of each element of x - if complex results is returned ‘ll be abort it

cbrt (x) %compute the real-valued cube root of each element of x - result will be negative if x is < 0

nthroot (x, n) %compute the real (non-complex) n-th root of x

abs (z) %compute the magnitude of z - |z|=sqrt(x^2 + y^2)

conj (z) %return the complex conjugate of z - conj(z)=x-iy

imag (z) %return the imaginary part of z as a real number

real (z) %return the real part of z

deg2rad (deg) %convert degrees to radians

rad2deg (rad) %convert radians to degrees

sin (x) %compute the sine for each element of x in radians - use ‘sind(x)’ for large numbers

cos (x) %compute the cosine for each element of x in radians - use ‘cosd(x)’ for large numbers

tan (x) %compute the tangent for each element of x in radians - use ‘tand(x)’ for large numbers

sec (x) %compute the secant for each element of x in radians

csc (x) %compute the cosecant for each element of x in radians

cot (x) %compute the cotangent for each element of x in radians

ceil (x) %return the smallest integer not less than x

fix (x) %truncate fractional portion of x and return the integer portion

floor (x) %return the largest integer not greater than x

round (x) %return the integer nearest to x

roundb (x) %return the integer nearest to x - if there are two nearest integers, return the even one

* 1. Array Operations *- valid for Vectors and Matrices*

cat (dim, a1, a2, …, aN) %return the concatenation of N-D array objects along dimension *dim*

horzcat (a1, a2, …, aN) %return the concatenation of N-D array objects along dimension 2

vertcat (a1, a2, …, aN) %return the concatenation of N-D array objects along dimension 1

reshape (a, m, n, …) %return a matrix with the specified dimensions (m, n, …), elements are taken from x

isinf (a) %return a logical array which is true where the elements of x are infinite and false if not

isnan (a) %return a logical array which is true where the elements of x are NaN values and false if not

isfinite (a) %return a logical array which is true where the elements of x are finite values and false if not

fliplr (a) %flip array left to right - return a copy of x with the order of the columns reversed

flipud (a) %flip array upside down - return a copy of x with the order of the rows reversed

sort (a, *dim, mode*) %return x arranged along dimension *dim* in *mode* order *- ascend or descend*

issorted (a, mode) %return true if the array is sorted according to mode *- "ascend", "descend", or "either"*

sum (a, *dim*) %sum of elements along dimension *dim*

prod (a, *dim*) %product of elements along dimension *dim*

sumsum (a, *dim*) %cumulative sum of elements along dimension *dim*

cumprod (a, *dim*) %cumulative product of elements along dimension *dim*

sumsq (a, *dim*) %sum of squares of elements along dimension *dim*

max (a, *dim*) %find maximum values

min (a, *dim*) %find minimum values in the array x

cummax (a, *dim*) %return the cumulative maximum values along dimension *dim*

cummin (a, *dim*) %return the cumulative minimum values along dimension *dim*

dot (a1, a2, *dim*) %compute the dot product of two vectors

cross (a1, a2, *dim*) %compute the vector cross product of two 3-dimensional vectors x and y

* 1. Vector Operations

any (v) %return True when any element of a vector is nonzero

all (v) %return True when all element of a vector is nonzero

flip (v) %return a copy of array x flipped

find(v==n) %returns the positions of the vector where the cond is True

* 1. Matrix Operations

any (mat, dim) %return row of bools*,* each elements indicates if any element of a *dim* is nonzero - True

all (mat, dim) %return row of bools*,* each elements indicates if all element of a *dim* is nonzero - True

flip (mat, dim) %return a copy of array x flipped across dimension dim - *2 for cols and 1 for rows*

sortrows(mat, c) %sort the rows of the matrix A according to the order of the columns specified in c=[idx]

pinv(mat) %matrix inverse

# 

# Data

* 1. load FILE %load the data
  2. Show variable

who

whos %and its details

* 1. Clean variable

clear VAR\_NAME %without var name, clean all of them

* 1. Save variable

save FILE.mat VAR

# Graphics

plot(x,y) %line plot

cd 'URL'; print -dpng Arq.png %create a file for the graph

hold on %allows you to overwrite graphics

Parameters

xlabel('Legenda') and ylabel('Legenda') %legend axis x and y

legend('Leg','Leg1',...,'Legn') %legend to the each rows

title('Graphic Title') %title

Figure

figure(1); %creates a figure for the plot (we can create n)

subplot(l,c,escolha) %creates a space for l\*c different plots and c chooses which position (all in a same figure)

clf %clean figure

1. .