① Scalar, positive, integers
② $0 < day < 32$, $0 < month < 13$) return
③ leap year, feb, → ≤ 29 → valid
>29 → Not valid) return

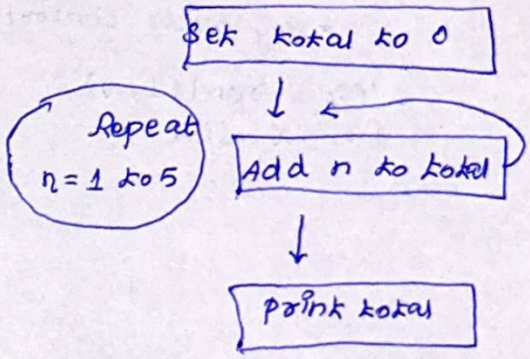
④ Non leap year → a = [31 30 31 30 31 30 31 31 30 31 30 31]

a [month] ≤ day → valid → return
↳ else → not valid.

Loops

n = 1:5 ; → matlab uses internally loop

sum( ) → loop is being used by matlab



[1 2 3 4 5]

total = 0
n = 1 total = 1
2 total = 3
3 total = 6
4 total = 10
5 total = 15
loop ends

MATLAB for loop

```
total = 0;
for n = 1:5
    total = total + n;
end
fprintf('total is %d \n', total);
```

Debugging : click the numbers

1
2 → Ready to debug
3 'placed a break'
4

Points

loop
```
for n = 1:5        → control statement
    total = total+n;   → body
end
```

using rand

```
rst = rand(1,5);
for x = rst
    if x > 0.5
        fprintf('Large\n');
```

```
    else
        fprintf ('Small \n');
    end
end
```

---

* Values assigned to the loop index don't have to be integers, regularly spaced or assigned in increasing order.

* They don't have to be scalars either (columns of array)

* Any other control statements can be used inside the body eg the for loop

```
list = rand(1, N)
for x = list
    :
    :
```

list = [0.8147, 0.9058, 0.1270, 0.9134, 0.6324]

x = 0.8147
x = 0.9058
x = 0.1270
x = 0.9134
x = 0.6324

x takes each element in each loop

>> rng(0) → we get same numbers for rand(1,5)

```
>> u = [5 4 8 8 2]
>> v = [5 5 7 8 8]
>> u - v  → matlab uses loop.
>> for x = 1 : length(u)
       w(x) = u(x) - v(x);
   end
>> w
w = [0  -1  -1  0  -6]
```

```
>> length(u) = 5
>> u(1) = 5
>> u(2) = 4
>> u(3) = 8
>> u(4) = 8
>> u[5) = 2
```

---

fibonacci

```
function f = fibo(n)
    if (~isscalar(n) || n < 1 || n ~= fix(n))
        fprintf ('n must be a +ve scalar integer \n');
    end
    f(1) = 0; f(2) = 1;
    for x = 3 : n
        f(x) = f(x-2) + f(x-1);
    end
end.
```

$P = A .* A$ → Same as $A .* A$

```
[row col] = size(A)

for r = 1 : row
    for c = 1 : col
      p(r,c) = A(r,c) * A(r,c)
    end
end
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .* \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\begin{array}{cc|cc} A_{11} * A_{11} & & A_{21} * A_{21} \\ A_{12} * A_{12} & & A_{22} * A_{22} \end{array}$$

Row 1 → c1, c2
Row 2 → c1, c2
) 2 for loop one inside other.

$$A = \begin{bmatrix} 9 & 10 & 3 & 10 \\ 10 & 7 & 6 & 2 \\ 2 & 1 & 10 & 10 \end{bmatrix}$$

```
for x = 1 : 3
    for y = 1 : x
        fprintf('*');
    end
    fprintf('\n');
end.
```

```
function pnv_tree(n)
for x = 1 : n
    for y = 0 : x-1
        fprintf(' ');
    end
    for z = n : -1 : x
        fprintf('*');
    end
    fprintf('\n');
end
end.
```

function halfsum → input matrix - A
→ o/p (summa)

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

1+2+3+5+6+9

1 → 1
2 → 2
3 → 3
(Col stacks with)

1, 2, 3, 4
2, 3, 4
3, 4
4

```
function summa = halfSum(A)
[row col] = size(A);
summa = 0;
for x = 1 : row
    for y = x : col
        summa = summa + A(x,y);
    end
end
end
```

while loops

"when we don't know no. of iterations"

eg: +ve integers add upto sum = 50.

"while loop"

```
while condition
   .
   .
   .
end
```

postsum.m

approx_sqrt.m

```
function y = approx_sqrt(x)
y = x;
while abs(y^2 - x) > 0.001 * x
    y = (x/y + y)/2;
end
end
```

root = 0.5 * (x + (N/x))

"Newton's method"

Busy: still working (may be ∞ loop)

"Try to avoid using i as variable"

↓ used as complex part

Next prime numbers?

```
function prime = next_prime(n)
n = n+1;
while (~isprime(n))
    n = n+1;
end
prime = n;
end
```

* %% → section
* Script has no arguments — uses workspace.

Examples

%% Example 1

% skipping is accomplished in the while condition.

```
ii = 1;
while ii < length(readings) &&
                readings(ii) <= 100
    readings(ii) = 0;
    ii = ii + 1;
end
```

'replace all elements of an array - until you see a 100+' value.

Name ← 'array_0_upto100'
(Script with sections) 'array_0_upto100_for' 'array_0_upto100_index'

%% Example 2

```
for ii = 1 : length(readings)
    if readings(ii) > 100
        break
    else
        readings(ii) = 0;
    end
end
```

%% Example 3

```
for ii = 1 : length(readings)
    if readings(ii) > 100
        break;
    end
end
fprintf('first reading above 100 is
        at index %d.\n', ii);
```

'Publish' — web version will be 'more' — html file to publish.

$$A = \begin{bmatrix} 81 & 10 & 50 & 15 \\ 90 & 28 & 97 & 12 \\ 91 & 2 & 4 & 10 \end{bmatrix}$$

```
>> size(a, 1)
ans = 1
>> size(a, 2)
ans = 20
```

Run script

```
>> A
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 97 & 12 \\ 91 & 2 & 4 & 10 \end{bmatrix}$$

→ when it sees 90+ skips & next step (on) turn to 0.

a = 1 × 20 matrix

```
for ii = 1:size(A,1)        → zoom in
    for jj = 1:size(A,2)
        if A(ii,jj) <= 90
            A(ii,jj) = 0;
        else
            break;
        end
    end
end
```

Logical Indexing

* Given a vector v, of scalars, create a second vector w that has only non-negative elements of v.

*

'Elegant Solution'

```
>> w = [] ; ii = 1;
>> u = [1, 2, 3]
>> w = [w, u(ii)]
   w = [1]
```

```
w = []; jj = 0;
for jj = 1 : length(v)
    if v(jj) >= 0
        jj = jj + 1;
        w(jj) = v(jj);
    end
end
```

Elegant

```
w = [];
for ii = 1 : length(v)
  if v(ii) >= 0
      w = [w  v(ii)];
  end
end
```

```
>> w = []
>> b = [1  2  3]
>> w = [w  b]
   w = [1  2  3]   → appending.
```

Much elegant - Ultimate solution.

'Logical Indexing'

```
w = v(v >= 0);   [Logical Indexing]
```

↳ Index based on logic.

```
function numfreeze = freezing(a)
numfreeze = length(a(a<32));
end.
```

```
>> [4  -1  7  5  3] > [5  -9  6  5  -3]
ans =
    0  1  1  0  1
```

```
>> c = [a>1 , a<1 , (3>2 && 4>5)]
c =
    1  0  1
```

```
>> holmes = logical([1  -2  0  0.9  0])
holmes =
    1  1  0  1  0
```

```
r = rand(10, 1, 6);
holmes = [1 1 0 0 1 1];
r(holmes)   → banned
```

'Index must be +ve integers / logic values'

```
>> a = [1 2 3 4];
>> a = a >= 0
   1  1  1  1
```

```
>> a = [1 0 1 0]
>> b = [2 3 4 5]
>> b(a) → can't be used
∴ a(0) → Invalid.
>> a(b) → Exceeds index
"can't be used"
```

valid

```
>> a = [1 0 1 0]
>> b = [2 2 2 2]
>> a(b)
   0  0  0  0
```

```
>> a = [1 0 1 0];
>> b = [2 2 2 2];
>> a (b > 2)
      ↳ [ ]
>> 1×0 empty double row vectors.
```

```
>> b = [2 3 4 5];
>> a (b > 2)
      ↓
   [0 1 1 1]
>> 0 1 0 (last 3 columns)
```

Note

"we can use logical arrays" as o/p

```
>> a = [1 0 1 0]
a = [1 0 1 0]
```

→ Not logical

"Even though it has 1's & 0's"

```
>> a = logical (a)
a =
   1×4 logical array
    1  0  1  0
```

```
>> b = [1 2 3 4];
>> b(a)
   1  3   → 1st column & 3rd column
```

Index 1 → Taken into account
Index 0 → eliminate

v (v > v0) → we get rid of elements that are not greater than the simultaneous element in v_0.

```
v  = [1 2 3 4 5]
v0 = [2 1 0 1 5]

v (v > v0)
   ↳ [0 1 1 1 0]

v([0 1 1 1 0]) → [2 3 4] → o/p
```

"Logical arrays can be given as index" - size (length) - must suit.

V = [1 2 -1 -2 3 4 5]

>> V(V<0) = 0

V = [1 2 0 0 3 4 5]

>> A = [1 2 3 ; 4 5 6]

A =

1 2 3

4 5 6

>> B = A(A>2)

B =

4
5 → why not 2×3
3 a column vector
6

[3
4 5 6] → Not a matrix

"Matlab gives up keeping a 2×3 array it saves as a column array"

A = [1 2 3
4 5 6]

A = A(:) → 1 2 3 4 5 6

"Exactly what happens in left side"

A(A>2) → 4 5 3 6

"Column vectors in column major order".

"1st Column - 1st"

"stability" - returns as column array.

>> stem(x, y)

t = 0 : 0.001 : 1;

y = 2 * cos (2 * pi * 1000 * t);

plot (t, y);

f = 1000

t = 0 : (0.001/1000) : (1/1000)

t = 0 : (0.001/100) : (1/100);

A = [1 2 3
4 5 6]

A(A>0) → 1 4 2 5 3 6

Column major order.

Set 5×5 array to 0

>> rng(0); A = randn(5)

A(A<0) = 0

A =

0.5377 0 . . . .

. . . . .

. . . .

5×5 → continues.

```
>> a = randn(5);
>> b = a(a<0)
b =
    :
```
Column vectors

```
>> a(a<0) = 0
```
5×5 matrix

'When a matrix: logic indexed right side: assigned to some other- major column wise transisition happens'

'Left side: a(a<0) = 0 = Assignment (Inner): No transition happens'
∴ 'Since size always remains as matrix which the previous case is not'

Right of '=' sign → Column transition
Left of '=' sign → No Column transition.

'Both sides' – Same matrix – No transition.

```
>> A(A>0.1) = sqrt(A(A>0.1))
```
5×5 matrix

$$A = \begin{bmatrix} 89 & 82 & 11 & 53 \\ 33 & 5 & 59 & 42 \end{bmatrix}$$

$$B = \begin{bmatrix} 34 & 44 & 52 & 64 \\ 62 & 73 & 58 & 99 \end{bmatrix}$$

```
>> A(A>B) = A(A>B) - B(A>B)
```
No transition — Transition happens

$$A>B = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$B>A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 55 & 38 & 11 & 53 \\ 33 & 5 & 1 & 42 \end{bmatrix}$$

$A(A>B) = 89, 82, 59$

$B(A>B) = 34, 44, 58$

$55 = 89 - 34$
$38 = 82 - 44$
$1 = 59 - 58$

A(A>B)
↳ logical array as index (same size).

Pre allocation

'Time a function'

Function : tic, toc [Stop watch]

```
>> tic;
>> sum(1:1000);
>> toc;
```

```
>> tic; sum(1,1000); toc
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 & \cdots \\ 2 & 4 & 6 & 8 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

```
>> function nopreallocc
N = 5000;
for ii = 1:N
    for jj = 1:N
        A(ii,jj) = ii*jj;
    end
end
```

```
>> tic; nopreallocc; toc
Elapsed time is 46.861254 seconds.
```

Small alteration

```
function nopreallocc
N = 5000;
A = zeros(N,N);
for i = 1:N
    for j = 1:N
        A(i,j) = i*j;
    end
end
```

```
>> tic; preallocc; toc
Elapsed time is 0.596797 seconds
```

'78 times faster'

why: In nopre allocation: matlab has no idea how big the matrix is!
Need to update every time in memory allocation

Case:2 - Already allocated

Just assign (update the value)

* 'Memory allocation - over & over' - Take time.
* Most of the time - unnecessary (Just initialize).

'Allocating before using (need) - pre allocation.

Preallocate every time - using large array - whose dimensions you know

'matlab gives the same advice' - Note the script carefully.

A → size change on every loop
(consider pre allocating)

loops - everywhere - without it - takes only billion years to solve problems.

max_sum → v (row vector), n (+ve integer)

→ n consecutive elements of v (whose sum is the possible largest sum)

v = [1 2 3 4 5 4 3 2 1], n = 3

∴ 4, 5, 4 = 13 (largest sum)

2

↓

1, 2, 3, 4, 5, 6, 7, 8 → length(v) - 1

9, (n=2)	9 (n=3), upto
↓	↓
8	7

'length(v) - n + 1'

9-2+1	9-3+1
8	7

n=2	n=3
x(1) + x(2)	x(1) + x(2) + x(3)
x(2) + x(3)	x(2) + x(3) + x(4)

∴ sum(1 : 1+n-1)

1 : 2

1 : 1+3
1 : 3
1 : n

Key: * Initial → 1 : n = summa

* In loop → i = (1 : length(v) - n + 1)

then

* v(1 : 1+n-1)

```
function [summa, index] = max_sum (v,n)
if n> length (v)
        index = -1; summa = 0; return;
end
        index = 1; summa = sum (v(1 : n));
        for i = 1 : (length (v) + 1 - n)
            if  sum (v(i : i+n -1)) >summa
                summa = sum (v(i : i+n -1));
                index = i;
            end
        end.
```

```
>> [summa , index] = max_sum ([1  2  3  4  5  4  3  2  1], 9)
summa = 25
index = 1
```

"Consecutive Sum" – maximum sum.

---

week-8 – Data types

Limitation of Computers:

Real numbers in mathematics:

* Can be infinitely large
* Have infinitely fine resolution – difference b/w two num can be only small

* "Computers – finite memory" – limit.
* memory can't store irrational numbers.
* π → appr of π
* $\sqrt{2}$ → appr of $\sqrt{2}$

no. of values represented by variable in matlab → limited

"True for any language"

Data types

*Set of values/operations that can be performed to those values.

* No. of dimensions
* Size in each dimensions
* Elementary type

→ matlab allows a fn function to be called with different data types – "strict rule": All elements of a fn array must be same type"

>> a=[1 2.2 3 4]

a = [1.0000 2.2000 3.0000 4.0000]

data type

double - default data type for all numerical values (matlab)

>> a=2

>> class(a)

double

>> class(sqrt(-1))

double.

>> whos

'prints all the variables in the current workspace'

>>class(a)
ans = double

Name	Size	bytes	Class
ans	1x6	12	char

★ All scalar → 1×1 matrix

★ String: row vector of 'char' data type

double → 64 bits (8 bytes) (Default)

12.34 = 1234 × 10^{-2}

'store 2 integers' → 1234, −2 to represent a non-integer 12.34

>> class('sup?')

ans =

char

>> class(4<3)

ans =

logical.

1234 → mantissa

−2 → exponent

★ 'matlab uses floating point representation for all doubles'

'uses 8 bytes = 64 bits'

single:

'other floating point type supported by MATLAB'

32-bit floating point

length → only 32 bits, double → 64 bits (That's name)

Double: Extra space allows for more digits: double precession.

Integers:

* Signed & unsigned.
* Sensor values: Read in computer: Typical int, Color eg pixels.

8, 16, 32, 64 bit long (supports)

Variants

more bits - wider range.

---

double → Inf, NaN [out of range values - also double type]

Inf → $\frac{1}{0}$ (or) any value greater than the largest value that can fit into double.

NAN → Not a number. [Invalid value $\frac{0}{0}$]

Useful functions

>> class(1)
double

>> isa(x, 'double') → true/false.
↓
Particular type.

Range check

intmax, intmin — maximum & minimum of integer types

realmax, realmin

max & min for the floating point type.

```
>> intmax
2147483647
>> intmin
-2147483648
>> realmin
2.2251e-308
>> realmax
1.7977e+308
```

Conversion

Name of function = name of desired data type

* int8(x) * uint32(x) * double(x) etc..

operators

* Arithmetic → Same data - no problem
  → mixed - mode → many rules & restrictions

```
>> a = uint32(2);
>> a - 1.54
ans = 0
uint32

>> a - 1.34
ans = uint32
      1
```

Relational data type: → Any data type

Result: Always logical

x = single(98.76)	n = int8(-16)	m = uint16(1234)	K = uint8(500)
x = 98.7600	n = -16	m = 1234	K = 255 ↓ max value = 255

K = uint8(256)	>> K = uint8(-1)	class(K)	'matlab converts within its fixed bound'
K = 255 ↓ max value	K = 0 ↓ unsigned	→ uint8	↓ Fits.

Nearby uint value - Fitted

```
>> K = uint8(2)
>> class(K)
uint8
```
→
```
K = 2
class(K)
double
```
→ The very moment we assign K, data type changed.

```
>> K = 'It makes sense'
>> K =
   It makes sense
>> class(K)
'char'
```

intmax('uint8') → 255

intmin('uint8') → 0

uint32 → 0 → −2147483648

double → $1.7977e^{+308}$

single → $1.1755e^{-38}$ (min)

```
>> a = 'hello';
>> uint32(a)
104  101  108  108  111
>> uint32('Hello')
72   101  108  108  111
```
→ ASCII

Strings

→ fprintf : String-vectors of char-s

→ char: numerical data type → Encoding, decoding scheme
→ ASCII scheme.

* ASCII — 1960 (7 bit long — 128 Characters)

* Many more schemes are developed.

Print_ascii.m — prints all ascii values

```
function Char_Codes
for g = 33:126
    fprintf('%s', Char(g));
end
fprintf('\n');
```

Same result

```
fprintf('%s', g);
```

↳ Converted implicitly '%s'

'matlab-gn a character & produces a pixel pattern - that symbolizes - something to us'

```
>> a = 'MATLAB for Smarties'
>> a(1)
M
>> a(1:6)
MATLAB
>> a(end-8:end)
ans = Smarties.
```

Compare arrays

```
>> a = 'Kernel'
>> b = 'Colonel'
>> a == b
```

'matrix dimensions must agree' — Not same size.

```
>> c = 'Matlab'
>> b = c(end:-1:1)
'Balkam'
```

```
>> c == b
[0 1 0 0 1 0] → logical.
```

'There must be an other way'

```
>> a = 'hello';
>> b = double(a);
```

vector

encrypt →

```
>> b = Char(b+3)
Khoor
```

↓ Decrypt

Char(b-3)

↓

Implicit Conversion then again to char

```
>> line1 = '1 2 3 4 5'
>> line2 = 'hello'
>> txt = [line1; line2]
>> txt'
```

```
1 2 3 4 5
h e l l o
```

```
1h
2e
3l
4l  5o
```

>> poem = ['Tiger', 'tiger', 'burning bright'; 'In the forest of the night']

Error using vertcat

Dimensions of arrays being concatenated are not consistent.

↓

we are trying to create a matrix of different lengths.

'vertically - catinate'

28 chars ; 27 chars

Not equal dimension.

Built in functions - String

* Searching for a substring → findstr
* Converts number to string → num2str
  vice versa → str2num
* strcmp → compares strings
* lower, upper → changes to caps / lower case.
* sprintf → fancy format → (into string)

>> findstr ('matlab', 'lab')

ans = 4.

>> findstr ('matlab', 'Lab')

ans = [ ]

'case sensitive'

>> mooc = 'MATLAB for smarties'

>> findstr (mooc, 'r')

ans = [10 15]

'Ignore case sensitivity'

>> strcmpi (mooc (1:6), 'matlab')

ans = 1

>> ppp = 3.1416;

>> pppp = '3.1416' } Not same

pppp = 3.1416 → 'MAILAB' doesn't use quotes.

'Numbers are intended'

'Strings are not'

>> strcmp (mooc, lang)

ans = 0 } why?

>> strcmp (mooc (1:6), lang)

↳ 'MATLAB'

ans = 1

↓

If Exact → true

else → 0

>> size (ppp) → number

1

>> size (pppp) → char.

1 × 6

>> ppp + 1 → 4.1416 (Implicit conversion)

» ppp + 1 (Ascii then + 1)

52 47 50 53 50 55 → Implicit Conversion

3 → [redacted] + 1 = 52.

» str2num (ppp)

3.1416

» ans + 1

52 47 50 53 50 55

» sprintf ('The area of a circle with radius %.2f is %.2f !\n', r, pi*r^2);

The area of a circle with radius 12.00 is 452.39!

» sprintf ('..... , r, pi*r^2);

→ Nothing in Command window

» str = sprintf ('... , r, pi*r^2);

» str

str =

The area of a circle with radius 12.00 is 452.39!

sprintf → can be taken as o/p argument (saved to a variable)

» class (str)

ans =

char

Caesar's cypher → Encryption algorithm

caesar → (vector, shift amount)

→ coded

space to ~ → 32 through 126

If we shift from space to space (1)

−1 (space to ~)

enc = 32:126 → 95

'ABCD' → 65, 66, 67, 68

str = str + 0; → [65 66 67 68]

str(x) − 31 + n > 95

(or)

str = str − 31;

① n → must be smaller than 95

95 − 2 → 94

95 + x + n

'Bound to 32 to 126' → 0 to 95 then add 31. 95

① Bound n to 1 to 95 → mod (n, 95)

② Bound Str(x)+n to 1 to 95 → 0 to 95 Str(x) = Str(x) + n

→ greater than 95

Str(x) = mod ((Str(x)+n), 95)

↳ less than 0

Str(x) = Str(x) + n + 95.

③ Str(x) = Str + 31

Str = char(str)

```
function coded = caesar(str, n)
str = str - 31;
for x = 1: length(str)
    while abs(n) > 95
        n = mod(n, 95);
    end
    if str(x) + n > 95
        str(x) = mod((str(x) + n), 95);
    elseif ((str(x) + n) >= 1 && (str(x) + n) <= 95)
        str(x) = str(x) + n;
    else
        str(x) = str(x) + n + 95;
    end
end
coded = char(str + 31);
end
```

→ caesar.m

caesar_sol.m

---

back = caesar(wrap, -96) → wrap = '2345' → '1234'

50 - 96 = [-46 -45 -44 -43]

[50 51 52 53]

[49 50 51 52]

mod(wrap - first, last - first + 1) + first

-46 - 32, 126 - 32 + 1 → 49

[50 51 52 53] , -96

first = double (' ');
last = first ('~');
txt = double (txt) + key;
txt = char (mod (txt - first, last - first + 1) + first);

* txt = (50 - 96) = -46
* txt - 32 = -78 → bound
* mod (-78, 95) → -78
* Again + 32

---

circshift - shifts the positions of elements circularly.

* [' ' : '~'] → shift 96 times backwards circularly.
* we can have own element.

Write index of gn vector.

function casaron = Circshif(v, n)
v1 = [' ' : '~'];
[~, v] = ismember(v, v1); → returns position matrix of v in v1
v1 = circshift (v1, -n);
↳ forward
casaron = v1(v)
end.

>> mod (8, 2)
0
>> mod (2, 8)
2

∴ Bound to 95 (∵ gn is 32 to 126)

→ ∴ gn - 31
→ mod (within 95, 95)
↳ Ans
→ mod (above 95, 95)
Some answer b/w 1 to 95.
∴
→ ans + 32 → conv to char.

no need to shoot n within 95.
mod ? →

```
function Codeen = Caesaen(V,n)
Codeen = V - 31 + n;
Codeen = char(mod(Codeen,95) + 31);
end.
```

'Caesaen_obj - myversion.m'

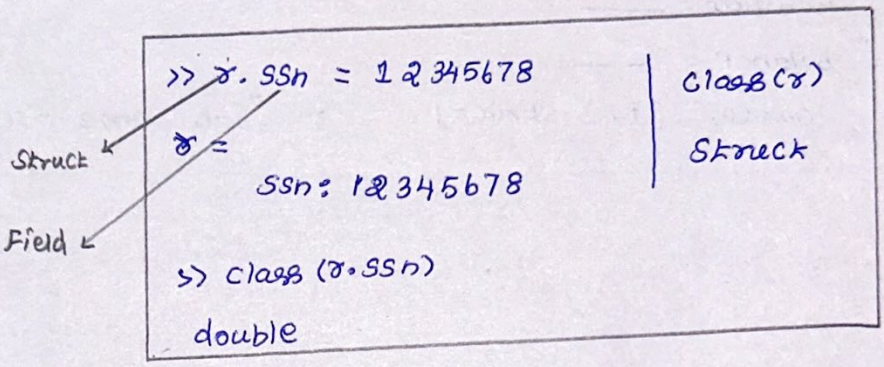
Struck

* homogeneous — array
* heterogeneous — Struck → fields (not element)

Access by field names, not indices

fields in the same struck can have diff types.

field of a struck can contain another struck




```
>> r.name = 'Homeen Simpson'
>> r =
    SSn: 12345678
    name: 'Homeen Simpson'
```

```
r.address.Street = '742 Evergreen Terrace'
>> r =
    SSn: 12345678
    name: 'Homeen Simpson'
    Address: [1x1 struct]
'struct inside struct'
```

arrays can hold structs

structs can hold arrays

* field of a struct can be arrays (homogeneity)

---

isfield → Is it has contain field

setfield → assigns a field (dynamic — while running)

rmfield → remove a field

struct → create a struct with certain fields.

'Bank account'

```
>> account.number = 1234;
>> account.balance = 5000;
>> account.owner.name = 'Hari'
```

```
>> account.owner.email = 'NIL'
>> class(account)
ans = struct.
```

>> account.owneen

ans

name = 'Hoori'

email = 'N91'

Array of Struct

>> account(2).numbeen = 123456

account =

1×2 Struct array with fields

numbeen

balance

owneen.

Struct array: same fields.

>> account(2).owneen

→ Not assigned yet.

ans = []

>> account(2)

numbeen = ——

balance = ——

owneen = [1×1 struct]

>> account(1).owneen.name

↓

can have different values

'But same fields'

>> length(account(1).owneen.name)

ans = 9

>> account(1).owneen.age = 23

>> account =

1×2 Struct with fields

numbeen

balance

owneen

→ only fields (enough)

one equal

↓

value can be different

∴ That's why owneen (acc(1)) has age

acc(2) → has no age.

>> account(2).owneen.age

'Not found'

'field inside field need not to be homogeneous'

>> account(1:2).owneen

ans =

name: 'Joe Smith'

email: 'joe@mat'

age : 23

ans =

name: 'Jane Foomeen'

owneen — Is the field need to be common.

'Not values'

↓

homogeneity.

Struck array: fields must be homogeneous
values can be different

>> isfield (account(1).owner, 'age')

ans = 1

>> isfield (account(2).owner, 'age')

ans = 0

>> rmfield (account(1).owner, 'age');

↳ can't change just changes
'Need to assign'

>> account(1).owner

ans =
name: 'Joe'
email: 'Joe'
age = 23

→ Not change
(Just returns version alone)

>> account(1).owner = rmfield (account(1).owner, 'age')

↳ Now age will be gone.

>> Course = struct ('Area', 'CS', 'number', 103, 'title', 'Intro to matlab');

Course =
Area: 'CS'
number: 103
title: Intro to matlab

] i/p arguments come in pairs

'Setting values for fields by processing a list values': (field: value pairs)

↓

Come from fortran (LISP)

↓

List processing
(AI - still used).

Cells -

'Store a page of text'

* 'Each line - seperate string':

* Array → Each row should be of same length
(Not a good way)

* Storing in a single string: not appealing

* Each line in seperate string: link each strings

'special objects' - pointers

* Idea: Vectors of special objects linking to each lines.

Pointers

* each variable (anything) - stored in memory - Address (index)
* consecutive address - neighbour cells
  * MATLAB calls a pointer a "cell"
  * Pointer (variable) – stores address

Cells

* MATLAB doesn't allow – to treat a cell as if it were a number
* strict rule – hard to make mistakes.
  * cell arrays – like structs – group heterogeneous data together (Index with numbers) – so used more frequently than structs.

Access Index

{}

```
>> page{1} = 'You could find'
>> whos
      workspace
>> class(page)
ans = cell.
```

```
>> class(page{1})
char
>> size(page{1})
ans =
     1    47
```

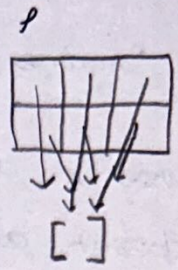
Pages

```
page{1} = 'Hi';
page{2} = 'Hello';
page{3} = 'Fine';
fprintf('\n');
for ii = 1:length(page)
    fprintf('%s\n', page{ii});
end
fprintf('\n');
```

→ display page (cell)

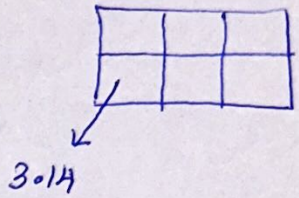
Cells are not just strings

```
>> P = Cell(2,3)

P =   []   []   []
      []   []   []
```



'Blank' - Each element has an address of the empty array.

```
>> P{2,1} = pi

P =   []      []   []
      [3.14]  []   []
```



* address is assigned - not - the value 3.14.
* 3.14 stored to some more - address stored in P{2,1}.

'Still P is homogeneous'

↓

All elements are cell type

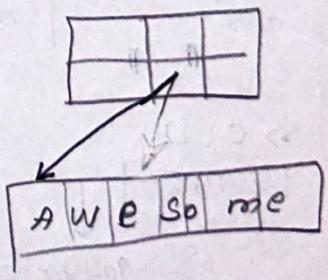
(even though it contains [] & scalars)

∴ It stores address alone : cell type

matlab cell array model allows individual elements to point at different types - but elements themselves are same data type (cell type)

'Address alone copied - not object itself'

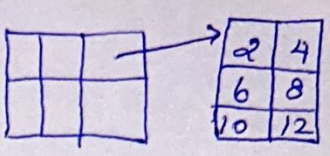
```
>> P{2,2} = 'Awesome'
```



* Pointed at first character of Awesome because the address used for vector/array/anything - is that its first element.

» P{2,3} = [2 4 ; 6 8 ; 10 12]

"Agains points first element"



Infact: makes no difference what element it points at, because MATLAB doesn't allow to look at

* numerical address
* can change it.

→ "strict control"

» P{1,3} = sum(P{2,3});

↓ store pointer.

↳ Right side: object need to be retrieved instead of address.

sum(P{2,3}); → [18 24] → Location is saved in P{1,3}

1/0 → Inf

» class(P{1,2}) → ans = double → cell object

» class(P(1,2)) → ans = cell → cell itself → matlab doesn't allow to see/change cell values.

» P{2,3}(3,2)

ans = 12

{2,3} → retrieves 2,3 object

$$\begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix}$$

then looks (3,2) ↓ 12.

C{1} = 1.4 , C{2} = 'Hello'

» C{1} → ans = 1.4

» C{2} → ans = Hello

» C → 1×2 cell array → {[12]} {'hello'}

» C(1) → [3.1456] → got brackets – looking at a pointer

» C(2) → 'Hello' → with quotes

→ look at pointers.

No quotes – string

quote – Not a string (cell)

* "Matlab doesn't allow two cells to point an object" (same)

```
>> c1 = { [1 2], [10, 20] }
>> c1 = [1x2 double] [1x2 double]
>> c2 = c1
c2 =
    [1x2 double] [1x2 double]
```

```
>> c1{1,1}
ans =
    1   2
>> c2{1,1}
    1   2
>> c1{1,2}
ans =
    10  20
>> c2{1,2}
ans =
    10  20
```

If same object: change at one position change others.

```
>> c1{1,1} = 'Straw'
c1 =
    'Straw'    [1x2 double]
>> c2{1,1}
ans =                    ⟶ No change.
    1   2
```

"It takes a copy in other cell - then assigns that copied values address in cell' — Not change the original.

"Cell pointers model: make copy - don't allow two cells point a single address."

c2 → points copy
c1 → points original
) not same address — don't change simultaneously.

↓

Not enforced by most languages C, C++, Java.

↓

Java more flexible.

"Matlab has two pointers → 1 (don't share two addres
→ 2 (like other languages)

Rarely used : Rarely needed for numeric problem solving.

cell → create cell

Sparse matrix → mostly zeros (matrix with most value zero)
↳ store where not zero alone (save space)

Sparse matrix

* cell vector – first 2 element – size
* second element – default value.

```
cell vec = {[2,3], 0, [1,2,3],[2  2  -3]};
```

```
function matrix = sparse2matrix(v)
row = v{1,1} (1,1);
col = v{1,1} (1,2);
def = v{1,2};
matrix = def * ones(row, col);
for x = 3:length(v)
    r = v{1,x} (1,1);
    c = v{1,x} (1,2);
    matrix (r,c) = v{1,x} (1,3)
end
end
```

```
b = v{1};
def = v{2};
matrix = def * ones(b);
for x = 3:length(v)
    r = v{x};
    matrix(r(1),r(2)) =
                r(3);
end
end.
```

Refer,

```
0   3   0
0  -3   0
```

```
>> a = {[2,3], 0,[1 2 3], [2  2  -3]};
>> b = a{1}
b =
    2   3
>> matrix = ones(b)
matrix =
    1   1   1
    1   1   1
>> c = a{3}
c =
    1   2   3
```

```
>> c(1)
   1
>> c(2)
   2
>> c(3)
   3
>> c = a(3)  → cell (not array)
{[1  2  3]}
```

we need array not cell

use curly braces

The String type - 2017 - intro

Problems: using chars to hold strings - matlab introduced some features.

```
>> fruit = 'Strawberries'
>> class (fruit)
ans = 'char'
>> size (fruit)
ans = 1   12
```

String - Informal way of denoting vector of characters

'Row vector'

→ up to 2014 version no single quotes

MATLAB - uses quotes now (2017)

```
>> a = 'Straw'
a =
'Straw'
```

other String type

```
>> a = "Hello"
a = "Hello"
```

```
>> class(a)
'string'
```

```
>> size (a)
ans =
   1   1   (not 1 x 5)
```

'See'

Any function: handles char & string as same.

```
>> fprintf ('Hello');
Hello
>> fprintf ("Hello");
Hello
```

→ No difference

'works similarly'

```
>> a = 'The rain in Spain';
>> strfind (a, 'in')
ans =
    7   10   16   27   32   42
>> b = "Bushy-headed trainee"
>> strfind (b, 'in')
ans =
    17   22   38
```

Every substring starts with a lowercase or followed by more lowercase letters

regexp → regular expression

```
>> regexp (b, 'a[a-z]+r')
```

a[a-z]+z

starting with a → lowercases → ending with z

Concept: function working fine with both char & string.
'polymorphic functions'

function out = my_str_function(in)
out = in(end:-1:1);

→ >> my_str_function('straw')
out = 'warts'

>> my_str_function(''straw'').
out = ''straw'' → No change

∴ >> size(''straw'')
ans = 1 x 1

>> my_str_function(char(''straw''))
→ switch back & forth
ans = 'warts'
>> String(ans)
''warts''

'' '' → '' (points)

'String doesn't fall into the limitations of char type'

>> a = [''ace''; ''king'']
a =
2 x 1 string array
''ace''
''king''
] Not same size incase of char

size(''ace'') → 1 x 1
size(''king'') → 1 x 1

>> a = ['ace'; 'king']
dimensions must be consistent
∴ size('ace') → 1x3
size('king') → 1x4
) Columns different

>> a = [1 2 3; 1 2 3 4]
'dimension error'
>> a = [''ace'', ''king'']
>> b = char(a).
2 x 4 char array
'ace' → empty space
'king'
) matlab using char adjusts to max no. of columns (putting space)